57个经典实例
3个综合项目案例

Elasticsearch
搜索引擎开发实战

罗刚 张子宪◎编著

深入剖析大规模分布式搜索引擎的实现原理，详解Elasticsearch开发搜索引擎的相关技术
涵盖大数据搜索引擎融合、自然语言处理与搜索引擎融合、Spring Boot与Vue.js前端融合等相关技术

详解多个搜索算法，每个算法都有广泛的应用前景
通过大量实例和综合案例手把手带领读者快速上手
书中的实例和综合案例大多来源于作者负责的实际项目

机械工业出版社
China Machine Press

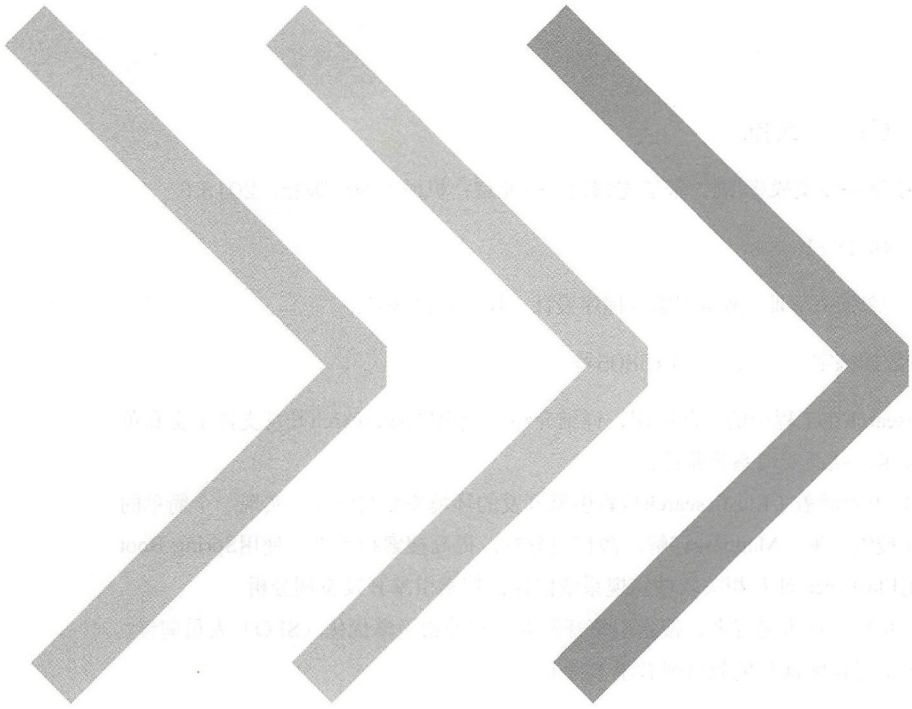
作者简介

罗刚

毕业于吉林大学。猎兔搜索创始人、IT培训讲师。曾经担任新东方创新研究院研究员，并担任首都师范大学研究生兼职讲师。创立猎兔搜索后带领团队先后开发出猎兔中文分词系统、猎兔信息提取系统、猎兔智能垂直搜索系统及互联网信息监测系统等，实现了互联网信息的采集、过滤、挖掘、搜索和实时监测。编写并出版了《自己动手写搜索引擎》《自己动手写网络爬虫》《使用C#开发搜索引擎》《网络爬虫全解析》等技术书籍。

张子宪

曾经在美国北乔治亚大学从事语言信息处理方面的研究和教学工作。现任职于聊城大学，从事自然语言处理的研究和教学工作，并从事机器翻译和计算机辅助翻译等领域的研究。在《中国科技论文》等核心期刊上发表过多篇论文。



Elasticsearch

搜索引擎开发实战

罗刚 张子宪◎编著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Elasticsearch搜索引擎开发实战/罗刚，张子宪编著. —北京：机械工业出版社，2018.6

ISBN 978-7-111-60348-1

Ⅰ. E… Ⅱ. ①罗… ②张… Ⅲ. 搜索引擎－程序设计 Ⅳ. ①TP391.3

中国版本图书馆CIP数据核字（2018）第145805号

本书结合Elasticsearch在工程中的实际应用，详细介绍了使用Elasticsearch开发支持中文和英文搜索引擎的相关技术，从而实现系统监控。

本书共分为8章，内容涵盖了Elasticsearch搜索引擎开发的环境安装与配置；实现一个简单的网站搜索；开发中文搜索引擎；Mapping详解；源代码分析；提高搜索相关性；使用Spring Boot开发搜索界面；使用Elasticsearch和相关软件实现系统监控；搜索引擎开发案例分析。

本书非常适合信息检索技术爱好者、搜索引擎开发人员和搜索引擎优化（SEO）人员阅读，也适合作为高等院校信息检索课程的教材或教学参考书。

Elasticsearch 搜索引擎开发实战

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）

责任编辑：欧振旭　李华君　　责任校对：姚志娟

印　　刷：中国电影出版社印刷厂　　版　　次：2018 年 7 月第 1 版第 1 次印刷

开　　本：186mm × 240mm　1/16　　印　　张：15.75

书　　号：ISBN 978-7-111-60348-1　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426　88361066　　投稿热线：（010）88379604

购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

前言

搜索引擎在人们的日常生活中发挥着越来越重要的作用。随着开源软件的普及与发展，涌现出了许多优秀的搜索软件，如 Elasticsearch、Solr 等。其中，Elasticsearch 以大规模分布式搜索见长，而 Solr 则以分面搜索见长。

本书选择 Elasticsearch 作为实现搜索引擎的工具。Elasticsearch 具有强大的分布式搜索和可视化功能，不仅丰富了实现搜索引擎的方法，而且还使复杂抽象的数据结构与算法变得直观而鲜活，因此在国外被迅速地引入到人工智能的相关课程中。

本书全面、系统地介绍了分布式搜索引擎的相关内容及 Elasticsearch 中的 Java 代码实现。本书内容既注重基础知识，又非常注重实践，每章都提供了大量的实例程序。读者可以通过这些实例快速上手，并迅速提高搜索引擎开发技术。通过对本书内容的学习，读者不仅可以掌握搜索引擎开发的基本知识，而且还可以灵活地将 Elasticsearch 运用到解决实际问题当中，从而提升工作效率。

本书特色

1. 内容全面，结构合理

本书首先介绍了 Elasticsearch 的安装和基本使用方法，然后介绍了从搜索到内容监控等方方面面的知识。在内容安排上，本书根据读者的认知规律对学习梯度做了合理安排，降低了学习难度。

2. 讲解详尽，实例丰富

本书对每个技术要点都做了细致入微的介绍，并且在讲解的过程中提供了丰富的实例，而且每个实例都经过精挑细选，具有很强的针对性，特别是本书最后的应用案例，更是对相关技术的一个全面应用。另外，书中所有实例的实现代码都考虑了通用性，读者可以直接将代码移植过来加以修改，即可解决自己的实际问题。

3. 语言通俗，图文并茂

本书用通俗易懂的语言进行讲解，尽量避免生疏的专业术语。在讲解一些重要知识点

时，书中给出了大量的图示及实例运行结果，帮助读者更加直观、高效地理解所学内容。

4．提供配套教学PPT，使学习更高效

为了便于读者高效、直观地学习本书内容，作者特意针对每章的重点内容制作了教学PPT，这些PPT和本书的实例源文件都会免费提供给读者下载。

本书内容

本书共分8章，具体内容介绍如下：

第1章Elasticsearch开发搜索引擎应用，主要介绍了搜索引擎开发方面的一些基础知识和Elasticsearch开发环境的安装，并对Java API与Elasticsearch搜索集群的交互也做了介绍。

第2章开发中文搜索引擎，主要介绍了中文搜索引擎开发的相关内容，包括中文分词原理和中文分词插件开发等。

第3章Mapping详解，主要介绍了Mapping概念及如何使用Mapping，包括Mapping索引、Mapping数据类型、Mapping参数和动态Mapping等。

第4章深入源码分析，详细分析了Elasticsearch源代码，主要内容包括Lucene源码分析、启动搜索服务、Guice框架、日期和时间库、Transport模块、线程池、模块、Netty通信框架、缓存、分布式、Zen发现机制、联合搜索和JVM字节码等。

第5章提高搜索相关性，主要介绍了向量空间检索模型、BM25检索模型、学习评分、查询意图识别和图像特征提升检索体验等内容。

第6章搜索界面开发，涵盖的主要内容包括使用Searchkit实现搜索界面；Spring Boot入门；Java模板引擎Pebble介绍；通过Spring-data-elasticsearch项目访问Elasticsearch；REST基本概念；使用Vue.js开发搜索界面；使用Vue.js Paginator插件实现翻页；实现搜索接口；Suggester搜索词提示；Word2vec挖掘相关搜索词；部署网站；使用Rust开发搜索界面等。

第7章Elastic栈系统监控，主要介绍了使用Elasticsearch和相关软件实现系统监控，包括管理Elasticsearch集群、Logstash数据处理工具、Filebeats文件收集器、消息过期、Kibana可视化平台、Flume日志收集系统、Kafka分布式流平台和Graylog日志管理平台等内容。

第8章案例分析，主要介绍了双语句对搜索、内容管理系统站内检索，以及使用Elasticsearch搜索公开的药物临床试验项目信息等几个案例。

本书读者对象

- 信息检索技术爱好者；

- 搜索引擎开发人员；
- 搜索引擎优化（SEO）人员；
- 从事算法研究的技术人员；
- 高等院校理工科专业的学生和老师。

本书配套资源及获取方式

为了方便读者高效学习，本书特意提供了以下配套资源：

- 本书配套教学 PPT；
- 本书源代码文件；
- 本书涉及的一些开发工具的安装包。

这些配套资源需要读者自行下载，请登录机械工业出版社华章公司的网站 www.hzbook.com，搜索到本书，然后在页面上的“资料下载”模块下载即可。

本书作者

本书由罗刚主笔编写，其他参与编写的人员有张子宪、沙芸、柳若边、崔智杰、石天盈、张继红、罗庭亮。

在此感谢我的家人、同事及所有在本书写作过程中提供过帮助的人！另外，本书在编写过程中参考了一些开源代码，在此对相关作者也一并表示感谢！

虽然我们对书中所述内容都尽量核实，并进行了多次校对，但由于写作时间仓促，加之作者水平所限，书中可能还存在疏漏和错误之处，恳请广大读者批评、指正。联系我们，请发电子邮件到 hzbook2017@163.com。

罗刚

于北京

目录

第 1 章　Elasticsearch 开发搜索引擎应用

信息时代，可供获取的数据大量涌现。那么如何通过搜索引擎从这些数据中挖掘出有价值的数据呢？正是基于这个需求，开源大数据搜索引擎 Elasticsearch 应运而生。

1.1　搜索引擎开发需求

网站搜索的一般需求有如下几点。

- 支持微服务：微服务架构模式可以用来构建复杂应用。
- 弹性负载：通过将搜索访问请求自动分发到多个服务节点上来扩展搜索系统对外的服务能力，实现应用程序容错。
- 容易部署：即集成的功能，不依赖第三方的分布式应用程序协调服务。
- 安全控制：控制非法的外部访问。
- 管理界面：管理搜索集群的健康状况，方便查看数据分布情况等。

1.2　准备开发环境

Elasticsearch 采用 Java 语言开发，所以我们需要先准备基本的 Java 开发工具 JDK，然后再准备运行在 JDK 上的 Eclipse。

1.2.1　Windows 命令行 cmd

假设有一个标准件工厂，在车间生产产品，在工地使用这些产品。与之类似，一般是在集成开发环境中开发软件，如果在 Windows 操作系统中运行开发的软件，则往往通过 Windows 命令行来运行。

在图形化用户界面出现之前，人们就是用命令行来操作计算机的。Windows 命令行是通过 Windows 系统目录下的 cmd.exe 程序执行的。执行这个程序最直接的方式是找到该程序，然后双击，但 cmd.exe 程序并没有桌面快捷启动图标，所以启动时比较麻烦。

鉴于此，可以在“开始”菜单的运行窗口中直接输入程序名，回车后运行这个程序。

具体操作方法：单击“开始”|“运行”命令，打开资源管理器中的运行程序窗口；或者直接使用快捷键——窗口键+R 键，打开运行程序窗口。然后输入程序名 cmd 后单击“确定”按钮，弹出命令提示窗口。因为可以通过这个黑屏的窗口直接输入相应命令来控制计算机，所以也称其为控制台窗口。

说明： Console，即控制台。遥控器上有控制面板，更复杂的设备往往有控制台。例如，一台机床或者数控设备的控制箱，通常会被称为控制台。顾名思义，控制台就是一个直接控制设备的台面，往往是一个上面有很多控制按钮的面板。在计算机里，把直接连接在计算机上的键盘和显示器叫做控制台。

通常用扩展名来表示文件的类别，如 exe 表示可执行文件。文件名称由文件名和扩展名组成，文件名和扩展名之间由小数点分隔，如 java.exe。

当我们建立或修改一个文件时，必须向 Windows 指明该文件的位置。文件的位置由三部分组成：驱动器、文件所在路径和文件名。路径是由一系列路径名组成的，这些路径名之间用“\”分开，如 C:\Program Files\Java\jdk1.8.0_03\bin\java.exe。

开始的路径一般是 C:\Users\Administrator，就像公园的地图上往往会标出游客的当前位置。Windows 命令行也有当前路径的概念，如 C:\Users\Administrator 就是当前路径。

可以用 cd 命令改变当前路径，例如，改变到 C:\Program Files\Java\ jdk1.8.0_03 路径，可以用如下命令：

```
C:\Users\Administrator>cd C:\Program Files\Java\jdk1.8.0_03
```

如果输入“cd d:”命令，这样的效果是改变当前路径到“d:”目录下。所以切换盘符不能使用 cd 命令，而是直接输入盘符的名称。例如想要切换到 D 盘，可以使用如下命令：

```
C:\Users\Administrator>d:
```

系统约定从指定的路径找可执行文件，这个路径通过 PATH 环境变量指定。环境变量是一个“变量名=变量值”的对应关系，每一个变量都有一个或者多个值与之对应。如果是多个值，则这些值之间用分号隔开。例如，PATH 环境变量可能对应这样的值：“C:\Windows\system32;C:\Windows”，表示 Windows 会从 C:\Windows\system32 和 C:\Windows 两个路径下寻找可执行文件。

设置或者修改环境变量的具体操作步骤是：首先在 Windows 桌面右击“我的电脑”，在弹出的快捷菜单中选择“属性”命令，在弹出的对话框中选择“高级”选项，然后在弹出的对话框中单击“环境变量”按钮，在弹出的对话框中设置用户变量或者系统变量，最后再设置环境变量 PATH 的值。

如果是用 Windows 7 以上的操作系统，可能找不到“我的电脑”快捷图标，其实打开桌面上“我的电脑”，就是运行资源管理器。打开资源管理器的另外一种方法是：按住键盘上的窗口键不放，然后再按 E 键之后选择“属性”标签，后面的操作相同，不再赘述。

环境变量设置完成后，需要重新启动命令行才能设置生效。为了检查环境变量是否已设置正确，可以在命令行中显示指定环境变量的值，需要用到 echo 命令。echo 命令用来

显示一段文字。例如：

```
C:\Users\Administrator>echo Hello
```

执行上面的命令后，将在命令行输出：

```
Hello
```

如果要引用环境变量的值，可以用前后两个百分号把变量名包围起来，如“%变量名%”。例如，使用 echo 命令显示环境变量 PATH 中的值：

```
C:\Users\Administrator>echo %PATH%
```

1.2.2　在 Windows 下使用 Java

本节首先介绍如何安装 JDK，然后介绍如何在命令行开发 Java 程序。Java 开发环境简称 JDK（Java Development Kit），JDK 包括 Java 运行环境（Java Runtime Envirnment）、一堆 Java 工具和 Java 基础类库。可以从 Java 官方网站 http://www.oracle.com/technetwork/java/index.html 下载得到 JDK，注意不是 http://www.java.com 下的 Java 虚拟机。

进入官网后，选择下载 Java SE，也就是 Java 的标准版本，然后选择 Latest Release 也就是最新发布的安装程序，完整的 JDK 版本号中包括大版本号和小版本号。例如 1.7.0 中的大版本号是 7，小版本号是 0，而 1.8.22 的大版本号是 8，小版本号是 22。因为可以在 Windows 或 Linux 等多种操作系统环境下开发 Java 程序，所以有多个操作系统的 JDK 版本可供选择。

因为 JDK 是有版权的，所以需要接受许可协议（Accept License Agreement）后才能下载。如果是在 Windows 环境下开发，就选择 Windows x86，这样会下载类似 jdk-8u121-windows-i586.exe 这样的文件，下载完毕后，使用默认方式安装 JDK 即可。

JDK 相关的文件都放在一个叫做 JAVA_HOME 的根目录下。JDK 根目录的命名格式是：C:\Program Files\Java\jdk1.8.0_<version>最后以一个数字类型的版本号结尾，如 10 或者 21 等。

因为一台计算机上可以安装多个 JDK 和 JVM，为了避免混乱，可以新增环境变量 JAVA_HOME，指定一个默认使用的 JDK。

使用 echo 命令检查环境变量 JAVA_HOME：

```
>echo %JAVA_HOME%
C:\Program Files\Java\jdk1.8.0_10
```

Eclipse 集成开发环境只需要 JAVA_HOME 这一个环境变量即可。如果要检查 JAVA_HOME 是否已经正确设置，使用如下命令后显示虚拟机的版本号就表示设置正确了。

```
>"%JAVA_HOME%"\bin\java -version
java version "1.8.0_10-rc"
Java(TM) SE Runtime Environment (build 1.8.0_10-rc-b28)
Java HotSpot(TM) Client VM (build 11.0-b15, mixed mode, sharing)
```

如果还需要在 Windows 控制台下执行 Java 程序，则需要访问编译源代码的 javac.exe 或者执行 class 文件字节码的 java.exe。环境变量 PATH 指定了从哪里找 java.exe 这样的可

执行文件，可以通过多个路径查找可执行文件，这些路径以分号隔开。如果想在命令行运行 Java 程序，还可以修改已有的环境变量 PATH，增加 Java 程序所在的路径。例如，C:\Program Files\Java\jdk1.8.0_10\bin。

然后检查环境变量 PATH：

```
>echo %PATH%
```

如果要检查 PATH 是否已经正确设置，只要在任何路径下输入 javac 命令都能显示 javac 的用法，就表示设置正确了，也可以用第一个 Java 程序试验一下。

新建一个 Java 项目后，在这个项目的 src 路径下新建一个叫做 Search 的 Java 类：

```
public class Search {
    public static void main (String args[]) {
        System.out.println("Hello Search!");
    }
}
```

运行结果如下：

```
>javac Search.java
>java Search
```

看运行结果是否显示 Hello Search!

最简单的方法是可以使用 javac 构建出 class 文件，对于复杂的项目，一般是使用工具构建项目源代码。Gradle 就是一个可用于构建 Java 项目的工具，Elasticsearch 本身也是使用 Gradle 构建。可以下载二进制文件来安装 Gradle，网址如下：

https://services.gradle.org/distributions/gradle-3.5-bin.zip。

在 Windows 上自动设置 Gradle 环境变量的脚本如下：

```
set input=F:\soft\gradle-3.5
echo gradle路径为%input%
set gradlePath=%input%
::创建GRADLE _HOME
wmic  ENVIRONMENT  create
name="GRADLE_HOME",username="<system>",VariableValue="%javaPath%"
call set xx=%Path%;%gradlePath%\bin
::echo %xx%
::将环境变量中的字符重新赋值到path中
wmic ENVIRONMENT where "name='Path' and username='<system>'" set
VariableValue="%xx%"
pause
```

打开控制台并运行 gradle -v 命令以显示版本来验证安装是否成功，例如：

```
C:\Users\Administrator>gradle -v
```

显示如下输出：

```
------------------------------------------------------------
Gradle 3.5
------------------------------------------------------------
Build time:   2017-04-10 13:37:25 UTC
Revision:     b762622a185d59ce0cfc9cbc6ab5dd22469e18a6
```

```
Groovy:      2.4.10
Ant:         Apache Ant(TM) version 1.9.6 compiled on June 29 2015
JVM:         1.8.0_121 (Oracle Corporation 25.121-b13)
OS:          Windows Server 2008 6.0 x86
```

可以在 Gradle 构建中使用标准和定制的 Ant 任务，就像在 Ant 自身中使用一样。另外，可以导入现有的 Ant 脚本，就像下面这样简单：

```
ant.importBuild 'build.xml'
```

1.2.3　Linux 终端

虽然使用 Linux 操作系统办公的人不多，但是很多大数据应用都运行在 Linux 操作系统下。

首先在 Windows 下安装 Chrome 浏览器，然后可以通过网址 http://sshy.us/登录 Linux 服务器。如果是用 root 账户登录，则终端提示符是“#”，否则终端提示符是“$”。

如果有现成的 Linux 服务器可用，可以使用支持 SSH 协议的终端仿真程序 SecureCRT 连接到远程 Linux 服务器上，因为可以保存登录密码，所以比较方便。除了 SecureCRT，还可以使用开源软件 PuTTY（http://www.chiark.greenend.org.uk/~sgtatham/putty），以及可以保存登录密码的 PuTTY Connection Manager。

使用 VMware、Linux 可以运行在 Windows 系统下，VMware 可以让 Linux 运行在虚拟机中，而且不会破坏原来的 Windows 操作系统。

首先要准备好 VMware，当然仍然需要 Linux 光盘文件。就好像华山派有剑宗和气宗，Linux 也有很多种版本，例如 RedHat、Ubuntu 及 SUSE，这里选择 CentOS（http://www.centos.org/）。

也可以在 Windows 下安装 Cygwin，使用它来练习 Linux 的常用命令。

如果需要安装软件，可以下载 RPM 安装包，然后使用 RPM 安装。但操作系统对应的 RPM 安装包找起来比较麻烦，一个软件包可能依赖其他的软件包，为了安装一个软件可能需要下载多个它所依赖的软件包。

为了简化安装操作，可以使用 Yum（Yellow dog Updater，Modified）来安装，也称其为黄狗升级管理器。Yum 会自动计算出程序之间的相互关联性，并且计算出完成软件包的安装需要哪些步骤，这样在安装软件时，不会再被那些关联性问题所困扰。

Yum 会自动从网络上下载并安装软件，有点类似于 360 软件管家，但是不会有商业倾向的推销软件。例如，安装支持 wget 和 rzsz 命令的软件有：

```
#yum install wget
#yum install lrzsz
```

可以使用 Nodepad++自带的插件 NppFTP 编辑 Linux 下的文件。有些生产环境的集群通过跳板机才能接触到。为了方便在服务器端管理和开发 Elasticsearch 相关应用，可以采用 Micro（https://github.com/zyedidia/micro）这样的终端文本编辑器。

可以使用 DNF 安装 Micro，在安装 DNF 前，必须先安装并启用 epel-release 依赖。使用 Yum 安装 epel-release 的命令如下：

```
# yum install epel-release
```

如果没有 DNF 安装工具软件，也可以直接安装 Micro 的预编译版本。使用 wget 下载 Micro：

```
#wget https://github.com/zyedidia/micro/releases/download/nightly/
micro-1.3.4-67-linux64.tar.gz
#tar -xf ./micro-1.3.4-67-linux64.tar.gz
```

编辑/etc/profile 配置文件，增加 Micro 所在的路径到 PATH 环境变量/home/soft/micro-1.3.4-67。

```
# ./micro /etc/profile
```

说明：和 Windows 不同，Linux 操作系统下的路径名之间用“/”分开。./micro-1.3.4-67-linux64.tar.gz 表示当前路径下的 micro-1.3.4-67-linux64.tar.gz 文件。

增加如下命令：

```
export PATH=/home/soft/micro-1.3.4-67:$PATH
```

可以使用 Micro 来编辑配置文件：

```
#micro  /etc/security/limits.conf
```

保存文件后，按 Ctrl+Q 组合键退出。很多 Linux 环境都带有 Python，如果版本太旧，读者可以自行安装。

```
# yum install  python34
```

下面先看下当前版本安装在了哪个目录下：

```
# which python
```

输出结果如下：

```
/usr/bin/python
```

一般使用 Bash 将用户可读的命令转换成计算机可理解的命令，并控制命令执行。

Bash 脚本中使用的特殊字符有：

```
#:Comments
~:home directory
```

在屏幕上打印“Hello”：

```
echo "Hello"
```

将 ABC 分配给 a：

```
a=ABC
```

输出 a 的值：

```
echo $a
```

在屏幕上打印 ABC。

将 ABC.log 分配给 b：

```
b=$a.log
```

输出 b 的值：

```
# echo $b
```

在屏幕上输出：

```
ABC.log
```

把文件 ABC.log 中的内容写入 testfile：

```
# cat $b > testfile
```

这里把 cat 命令的输出重定向到 testfile。

我们可以把重复执行的 Shell 脚本写入一个文本文件中。和 Windows 不同，Linux 不以文件后缀名作为系统识别文件类型的依据，但是可以作为我们识别文件的依据，因此我们可以将脚本文件以.sh 结尾。

可以使用 Micro 创建一个类似 script.sh 的文件 micro script.sh，创建好脚本文件后就可以在文件内用脚本语言要求的格式编写脚本程序了。此外，还可以使用 touch 命令先创建一个空文件 touch script.sh。

在创建的脚本文件中输入以下代码并保存退出。

```
#! /bin/bash
echo "hello world!"
```

然后添加脚本文件的可执行运行权限：

```
# chmod +x script.sh
```

运行文件./script.sh，结果如下：

```
hello world!
```

Shell 脚本中用“#”表示注释，相当于 C 语言的“//”注释。但如果“#”位于第一行开头并且是“#!”（称为 Shebang）则例外，它表示该脚本使用后面指定的解释器/bin/sh 解释执行。每个脚本程序必须在开头包含 Shebang 语句。

例如，使用参数 n 检查语法错误：

```
# bash -n ./test.sh
```

如果 Shell 脚本中有语法错误，则会提示错误所在行；否则不输出任何信息。

智能系统需要根据不同的外部情况做出不同的处理，所以需要使用流程控制语句。下面简单介绍一下 if 和 case 语句。

if 语句的语法如下：

```
if [ condition ] then
   command1
```

elif#和 else if 等价：

```
    then
        command2
    else
    default-command
fi
```

说明：这里的 fi 是 if 反过来写的。

例如，为了判断某个命令是否存在，可以使用以下格式：

```
if which programname >/dev/null; then
    echo exists
else
    echo does not exist
fi
```

如判断 Yum 是否存在：

```
if which yum >/dev/null; then
    echo "exists"
else
    echo "does not exist"
fi
```

case 语句的语法如下：

```
case 字符串 in
    模式 1)
        语句
        ;;
    模式 2)
         语句
         ;;
    *)
         默认执行的语句
         ;;
esac
```

这里的 esac 就是 case 反过来写。例如：

```
extension="png"
case "$extension" in
  "jpg"|"jpeg")
    echo "It's image with jpeg extension."
  ;;
  "png")
    echo "It's image with png extension."
  ;;
  "gif")
    echo "Oh, it's a giphy!"
  ;;
  *)
    echo "Woops! It's not image!"
  ;;
esac
```

这里使用“|”把 jpg 和 jpeg 这两个模式连接到了一起。

1.2.4　在 Linux 下使用 Java

本节首先安装 JDK，然后介绍如何在 Linux 终端开发 Java 程序。

使用 wget 下载 JDK 安装包：

```
#wget -c --header "Cookie: oraclelicense=accept-securebackup-cookie"
http://download.oracle.com/otn-pub/java/jdk/8u131-b11/d54c1d3a095b4ff2b
6607d096fa80163/jdk-8u131-linux-x64.rpm
```

然后使用 RPM 安装 JDK，命令如下：

```
# rpm -i ./jdk-8u131-linux-x64.rpm
```

验证 Java 安装是否成功，输入如下命令：

```
#java -version
```

如果安装成功，则输出如下结果：

```
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
```

为了自动构建 Java 源代码，需要安装 Maven。首先下载 Maven 安装文件：

```
# wget
 http://mirrors.shuosc.org/apache/maven/maven-3/3.5.2/binaries/
 apache-maven-3.5.2-bin.tar.gz
```

然后解压安装文件：

```
# tar -xzf apache-maven-3.5.2-bin.tar.gz
```

把安装路径改成/usr/local/apache-maven：

```
# mv apache-maven-3.5.2 /usr/local/apache-maven
```

修改配置文件/etc/profile 设定变量 MAVEN_HOME 的值，并把 mvn 所在的路径加入到 PATH 变量中，也就是增加如下命令行：

```
export MAVEN_HOME=/usr/local/apache-maven
export PATH=$MAVEN_HOME/bin:$PATH
```

然后安装 Gradle。首先下载安装文件 gradle-3.5-bin.zip：

```
# wget https://services.gradle.org/distributions/gradle-3.5-bin.zip
```

创建 Gradle 软件存放的路径：

```
# mkdir /opt/gradle
```

解压缩 gradle-3.5-bin.zip 到/opt/gradle 目录下：

```
# unzip -d /opt/gradle gradle-3.5-bin.zip
```

检查解压缩出来的文件：

```
# ls /opt/gradle/gradle-3.5
LICENSE  NOTICE  bin  getting-started.html  init.d  lib  media
```

把 gradle 所在的路径加入到 PATH 变量中：

```
export PATH=$PATH:/opt/gradle/gradle-3.5/bin
```

在 Linux 终端输入以下命令验证是否成功安装：

```
# gradle -v
```

1.2.5 Eclipse 集成开发环境

就像做实验有专门的试验台，开发软件也有专门的集成开发环境。开发 Java 程序最流行的工具是 Eclipse（网址是 http://www.eclipse.org）。

Eclipse 也有很多版本，可以选择最简单的一个版本 Eclipse IDE for Java Developers。Eclipse 是绿色软件，无须安装，解压后就可以直接使用，在 Windows 下，双击后就可以解压文件。如果需要专门的解压软件，推荐使用 7z（网址是 http://www.7-zip.org/）。

Eclipse 默认是英文界面，如果读者习惯用中文界面的话可以从这个网站 http://www.eclipse.org/babel/downloads.php 中下载支持中文的语言包。

Eclipse 把软件按项目进行管理，每个项目都有自己的.classpath 文件，指定了源代码路径，编译后将输出文件的路径及该项目引用的 jar 包的路径。一个简单的.classpath 文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
<classpathentry kind="src" path="src"/>
<classpathentry kind="src" path="test"/>
<classpathentry kind="con"

path="org.eclipse.jdt.launching.JRE_CONTAINER/org.eclipse.jdt.internal.
debug.ui.launcher.StandardVMType/JavaSE-1.8"/>
<classpathentry kind="lib" path="lib/fastjson-1.2.7.jar"/>
<classpathentry kind="output" path="bin"/>
</classpath>
```

为了方便在其他计算机上正常开发，classpathentry 中的路径一般使用相对路径而不是绝对路径。如果是绝对路径，也可以将文件路径手动修改为相对路径。

安装 Eclipse 的 Gradle 插件。首先从 https://github.com/eclipse/buildship 网站中找到安装地址，然后把文件解压缩到 eclipse\dropins 目录下就可以了；也可以在 Eclipse 界面上安装，选择菜单栏的“帮助”|“安装新软件”命令，然后输入插件地址即可。

1.3 了解 Elasticsearch

Elasticsearch 把输入文档和复杂的查询语法及输出的查询结果都封装成了 XContent，

这样数据就可以采用 XML 或者 JSON 格式表示成可读的形式。JSON 表示形式更简短，所以 Elasticsearch 采用 JSON 格式来表示 XContent。因为要使用 JSON 和 Elasticsearch 服务端打交道，所以本节将介绍 JSON。

1.3.1　JSON 数据格式

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，不仅易于人们阅读和编写，而且也易于计算机解析和生成，可以用它传输由名称/值对和数组数据类型组成的数据对象。一些结构复杂的数据也可以采用 JSON 格式来表示，如散列表中的值就是数组。

JSON 的基本数据类型介绍如下。

- 数字：有符号的十进制数字，可能包含小数部分，也可能使用指数 E 表示法，但不能包括非数字，如 NaN。该格式不区分整数和浮点数。
- 字符串：0 个或多个 Unicode 字符的序列。字符串用双引号分隔，并支持反斜杠转义语法。
- 布尔值：为 true 或 false 的任一值。
- 数组：0 个或多个值的有序列表，每个值可以是任何类型。数组使用方括号符号，元素以逗号分隔。
- 对象：名称/值对的无序集合，其中名称（也称为键）是字符串。由于对象旨在表示关联数组，推荐每个键在对象内是唯一的。对象用大括号分隔，并使用逗号分隔每对，而在每一对中，用冒号（：）将键或名称与其值分隔开。
- null：一个空值，使用单词 null。

一个表示 Elasticsearch 版本的对象如下：

```
"version" : {
  "number" : "5.3.0",
  "build_hash" : "3adb13b",
  "build_date" : "2017-03-23T03:31:50.652Z",
  "build_snapshot" : false,
  "lucene_version" : "6.4.1"
}
```

可以使用 Elasticsearch 提供的 API 构建 JSON 串。

例如，在 Eclipse 中创建一个 Gradle 项目，首先引入 jackson 相关的 jar 包，然后在 build.gradle 文件中增加依赖库：

```
runtime group: 'org.elasticsearch', name: 'elasticsearch', version: '5.6.2'
```

最后运行如下代码：

```
XContentBuilder b = XContentFactory.jsonBuilder().startObject();
b.field("title", "新闻标题");
b.field("body", "内容");
```

```
b.endObject();
// 从 XContent 到 JSON
String json = b.bytes().utf8ToString();
System.out.println(json);
```

输出结果如下：

```
{"title":"新闻标题","body":"内容"}
```

1.3.2 Elasticsearch 基本概念

Lucene 是由一个 Java 语言开发的开源全文检索引擎工具包。把 Lucene 用 Netty 封装成服务，使用 JSON 访问就是 Elasticsearch。

Elasticsearch 内置了对分布式集群和分布式索引的管理，所以相对 Solr 来说，不需要额外安装 ZooKeeper，其更容易分布式部署。使用 Elasticsearch 的搜索系统整体架构图如图 1-1 所示。

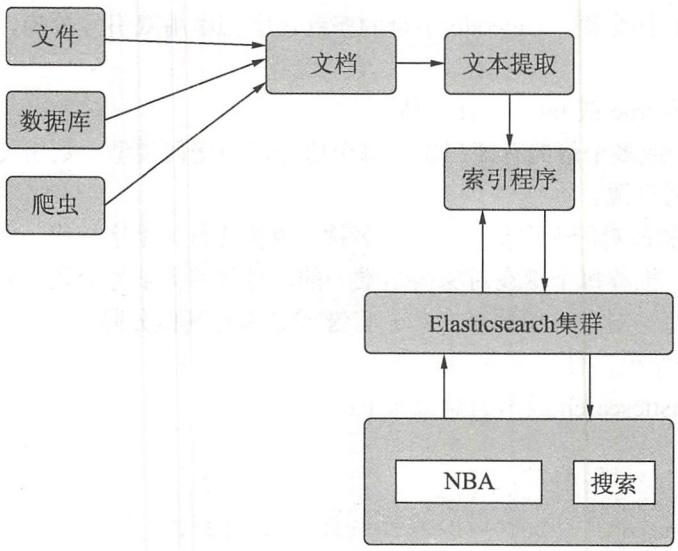


图 1-1　Elasticsearch 的外部结构

Elasticsearch 的每一个运行实例称为一个节点，既可以在同一台计算机上运行多个实例，也可以在每台计算机上只运行一个实例。

在一个分布式系统里，多个 Elasticsearch 运行实例可以组成一个集群（cluster），该集群里有一个动态选举出来的主节点（master）。如果主节点失败，会自动选出新的节点作为主节点，所以不存在单点故障。

在同一个子网内，只需要在每个节点上设置相同的集群名，这些集群名相同的节点会自动组成一个集群。Elasticsearch 包含了节点和节点之间通信模块及节点之间的数据分配和平衡模块。

为了实现容错，Elasticsearch 会把查询文档集合分解为多个小的索引，每一个小的索引就叫做分片（shards）。每一个分片都可以有 0 到多个副本（replicas），而每一个副本也都是分片的完整复制品，这样也提高了查询速度。

一旦 Elasticsearch 的某个节点数据损坏或服务不可用的时候，就可以用其他节点来代替坏掉的节点，以达到高可用的目的。当有节点加入或退出时，主节点会根据机器的负载对索引分片进行重新分配，当“挂掉”的节点再次重新启动的时候也会进行数据恢复（recovery）。

Elasticsearch 通过网关（Gateway）来管理集群恢复，可以配置群集需要加入多少个节点才能启动恢复数据。网关配置用于恢复任何失败的索引。当节点崩溃并重新启动时，Elasticsearch 将从网关读取所有的索引和元数据。

Transport 代表 Elasticsearch 内部的节点或者集群与客户端之间的交互方式，默认使用 TCP 协议进行交互，同时支持 HTTP 协议（JSON 格式）、thrift、Servlet、Memcached、ZeroMQ 等多种的传输协议（通过插件方式集成）。

为了让集群在运行时动态附加额外的功能，可以使用插件机制加载实现公共接口的程序集。Elasticsearch 插件用于以各种特定的方式扩展基本的 Elasticsearch 功能。

1.3.3　HTTP 协议

客户端通过 HTTP 协议和 Elasticsearch 服务器打交道。客户端发起一个到服务器上指定端口的 HTTP 请求，服务器端按指定格式返回网页或者其他网络资源，如图 1-2 所示。

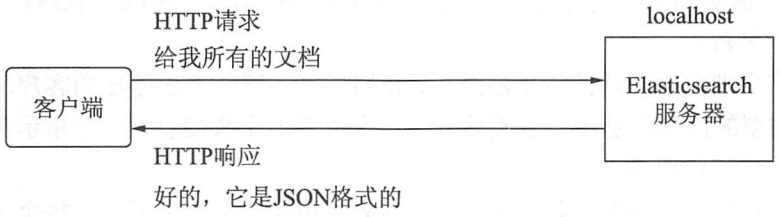


图 1-2　HTTP 协议示意图

就像发快递需要收件人的地址一样，打开网页也需要知道网络资源的地址。URI 包括 URL 和 URN，但是 URN 并不常用，也很少有人知道 URN。URL 由 3 部分组成，如图 1-3 所示。

需要使用 DNS 把主机名转换成 IP 地址，如果没有配置 DNS 则不能根据域名打开网站。

HTTP 协议传输的内容一般是超文本，但也可以是图像等，所以还需要头信息来描述内容的格式等信息。为了容易理解，协议头使用文本描述而不是二进制格式。

客户端向服务器发送的请求头包含请求的方法、URL、协议版本及请求修饰符、客户

信息和内容等。服务器以一个状态行作为响应，相应的内容包括消息协议的版本、成功或者错误编码、服务器信息、实体元信息及实体内容等。

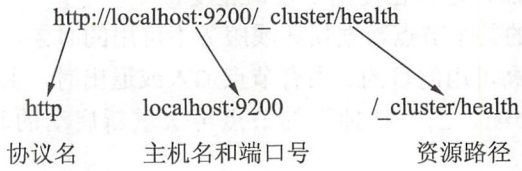


图 1-3　URL 分为 3 部分

HTTP 请求格式如下：

```
<request line>
<headers>
<blank line>
[<request-body>]
```

在 HTTP 请求中，第一行必须是一个请求行（request line），用来说明请求类型、要访问的资源及使用的 HTTP 版本；紧接着是头信息（header），用来说明服务器要使用的附加信息。头信息之后是一个空行，在此之后可以添加任意的数据，这些附加的数据称为主体（body）。

HTTP 规范定义了 8 种可能的请求方法。客户端经常用到 GET 和 POST，分别说明如下。

- GET：检索 URI 中标识资源的一个简单请求。例如，客户端发送请求 GET/_cluster/health HTTP/1.1。
- POST：服务器接收被写入客户端输出流中的数据请求，可以用 POST 方法来提交查询词等参数。

介绍完客户端向服务器的请求消息后，我们再来了解一下服务器向客户端返回的响应消息。这种类型的消息也是由一个起始行、一个或者多个头信息、一个指示头信息结束的空行和可选的消息体组成。

HTTP 的头信息包括通用头、请求头、响应头和实体头 4 个部分，每个头信息由一个域名、冒号（:）和域值 3 部分组成。域名与大小写无关，域值前可以添加任何数量的空格符，头信息可以被扩展为多行，在每行开始处使用至少一个空格或制表符，如图 1-4 所示。

例如，客户端发出 GET 请求：

```
GET /_cluster/health HTTP/1.1
```

服务器返回响应：

```
HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked
```

```
{
    "active_primary_shards": 0,
    "active_shards": 0,
    "active_shards_percent_as_number": 100.0,
    "cluster_name": "es-catalog",
    "delayed_unassigned_shards": 0,
    "initializing_shards": 0,
    "number_of_data_nodes": 3,
    "number_of_in_flight_fetch": 0,
    "number_of_nodes": 3,
    "number_of_pending_tasks": 0,
    "relocating_shards": 0,
    "status": "green",
    "task_max_waiting_in_queue_millis": 0,
    "timed_out": false,
    "unassigned_shards": 0
}
```

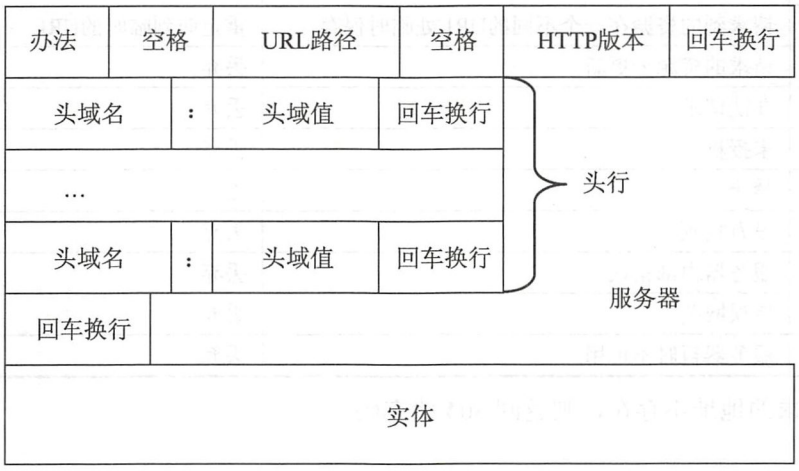


图 1-4　HTTP 请求信息格式

在服务器返回的响应中，第一行中的 200 就是一个状态码，状态码是一个 3 位数字的结果代码，爬虫可以用状态码识别 Elasticsearch 服务器处理的情况。状态码的第一位数字定义响应的类别，后两位数字有分类的作用。例如：

- 1xx，信息响应类，表示接收到请求并且继续处理。
- 2xx，处理成功响应类，表示动作被成功接收、理解和接受。
- 3xx，重定向响应类，为了完成指定的动作，必须接受进一步处理。
- 4xx，客户端错误，客户请求包含语法错误或者不能正确执行。
- 5xx，服务端错误，服务器不能正确执行一个正确的请求。

完整的状态码如表 1-1 所示。

表 1-1 HTTP常用状态码

状态代码	代码描述	处理方式
200	请求成功	获得响应的内容，进行处理
201	请求完成，结果是创建了新资源，新创建资源的URI可在响应的实体中得到	爬虫中不会遇到
202	请求被接受，但处理尚未完成	阻塞等待
204	服务器端已经实现了请求，但是没有返回新的信息。如果客户是用户代理，则无须为此更新自身的文档视图	丢弃
300	该状态码不被HTTP/1.0的应用程序直接使用，只是作为3XX类型回应的默认解释。存在多个可用的被请求资源	若程序中能够处理，则进行进一步处理，如果程序中不能处理，则丢弃
301	请求到的资源都会分配一个永久的URL，这样以后就可以通过该URL来访问此资源	重定向到分配的URL
302	请求到的资源在一个不同的URL处临时保存	重定向到临时的URL
304	请求的资源未更新	丢弃
400	非法请求	丢弃
401	未授权	丢弃
403	禁止	丢弃
404	没有找到	丢弃
500	服务器内部错误	丢弃
502	错误网关	丢弃
503	服务器暂时不可用	丢弃

如果请求的地址不存在，则返回 404 状态码。

1.4 Elasticsearch 安装和配置

本节首先介绍如何在 Windows 和 Linux 系统下安装 Elasticsearch，然后介绍将 Elasticsearch 作为一个系统服务自动启动的方法。

1.4.1 安装 Elasticsearch

若在 Windows 系统下安装 Elasticsearch，可以从网址 http://www.elasticsearch.org/download/上下载安装包。这里使用的版本为 5.1.2，得到的下载文件是 elasticsearch- 5.1.2.zip。

然后将下载的文件解压至某个目录下，如 D:\elasticsearch-5.1.2。解压后的文件中，bin 是运行的脚本，config 是设置文件，lib 中放依赖的包。到目录 D:\elasticsearch-5.1.2\bin 下，

运行 elasticsearch.bat。

如果显示 Java 虚拟机内存不够，则可以在 D:\elasticsearch-5.1.2\config\jvm.options 配置文件中调整内存大小。其中，Xms 参数表示堆空间的初始值，Xmx 参数表示堆空间的最大值，应该把最小和最大 JVM 堆设置成相同的值。例如：

```
-Xms2g
-Xmx2g
```

成功启动 Elasticsearch 后，在浏览器中输入网址 http://localhost:9200/。启动成功后，会在解压目录下增加两个文件夹：data 文件夹件用于存储索引数据，logs 文件夹用于日志记录。因为创建索引较耗时，所以文档会被预先写入到一个日志目录中。

用户通过 HTTP 协议发送指令和 Elasticsearch 交互，可以用命令行工具 CURL 发送 GET 或者 POST 命令与 Elasticsearch 打交道。Linux 默认已经安装了 CURL 命令行工具，但是也有 Windows 版本的 CURL，可以从网站 http://www.paehl.com/open_source/上下载一个编译好的 curl.exe 文件，然后在 Windows 命令行下使用 cmder 命令行工具运行这个工具。

默认情况下 Elasticsearch 的 RESTful 服务只有本机才能访问，也就是说无法从主机访问虚拟机中的服务。为了方便调试，可以修改 config/elasticsearch.yml 文件，加入以下两行命令：

```
http.host: 0.0.0.0
transport.host: 127.0.0.1
echo >> ./elasticsearch.yml http.host: 0.0.0.0
echo >> ./elasticsearch.yml transport.host: 127.0.0.1
```

但线上环境切忌不要这样配置，否则任何人都可以通过这个接口修改 Elasticsearch 中的数据。

为了能看到索引内容，需要安装 head 插件。Elasticsearch 5.x 安装 head 插件需要随同 Node.js 一起安装包管理工具 npm。下面介绍在 Linux 下的安装方法，首先安装 JDK。

使用默认环境启动 Elasticsearch 时可能会出现一些错误，所以需要进行一些设置。准备好 Elasticsearch 所需要的 Linux 操作系统环境，根据需要增加打开文件和进程的数量及虚拟内存数量。

编辑 limits.conf 文件：

```
# vi /etc/security/limits.conf
```

添加如下内容：

```
* soft nofile 65536
* hard nofile 131072
* soft nproc 2048
* hard nproc 4096
```

增大进程数的限制。然后编辑 CentOS 操作系统中的配置文件 90-nproc.conf：

```
# vi /etc/security/limits.d/90-nproc.conf
```

将如下内容：

```
* soft nproc 1024
```

修改为：

```
* soft nproc 2048
```

增加虚拟内存空间大小。修改配置文件 sysctl.conf：

```
# vi /etc/sysctl.conf
```

添加下面配置：

```
vm.max_map_count=655360
```

然后执行命令：

```
sysctl -p
```

在 Linux 下不能以 root 用户启动 Elasticsearch，所以需要先创建用户，这里创建一个名为 ops 的用户。

```
# adduser ops
```

设置密码：

```
# passwd ops
```

操作系统环境准备好之后，就可以安装 Elasticsearch 了。首先下载并解压缩安装包 elasticsearch-5.6.2.tar.gz。

```
$ wget https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-
5.6.2.tar.gz
$ tar -xvf elasticsearch-5.6.2.tar.gz
```

然后执行脚本启动服务进程：

```
# sh elasticsearch
```

这样会在终端以交互方式执行 Elasticsearch 服务进程。如果想让这个进程脱离终端运行，加上参数 -d 即可（./elasticsearch –d）。

对于旧版本的 Linux，启动时会提示警告：unable to install syscall filter。因为 Linux 内核不支持 secomp，导致与安全相关的过滤器安装失败。Elasticsearch 默认尝试使用 secomp，因此必须迁移到支持 secomp 的内核，或者禁用 bootstrap.system_call_filter。可以忽略这个警告。

可以使用 Linux 下的 pgrep 命令判断程序是否正在运行：

```
# pgrep java
```

或者使用 Java 提供的 JPS 工具进行察看。

```
# jps
24276 Jps
1113 Bootstrap
24701 Elasticsearch
```

也可以查看 logs 目录下的启动日志：

```
$ cat ./elasticsearch.log
```

客户端通过 HTTP 请求与 Elasticsearch 打交道，HTTP 请求包括请求的 URL 地址和

HTTP 命令（GET、POST）等。为了简洁而一致地描述 HTTP 请求，Elasticsearch 文档使用 CURL 命令行语法，这也是在用户社区中对 Elasticsearch 请求的标准做法的描述。例如，通过 CURL 命令给本地节点发送 HTTP 请求：

```
# curl -XGET 'http://localhost:9200/'
```

使用 CURL 命令行的简单搜索请求：

```
# curl "-XPOST" "http://localhost:9200/_search" -d'
  {
    "query": {
      "match_all": {}
    }
  }'
```

当上述代码片段在控制台中执行时，使用 3 个参数运行 CURL 程序。第 1 个参数-XPOST 意味着 CURL 所做的请求应该使用 HTTP 的 POST 请求；第 2 个参数“http://localhost:9200/_search”是请求的 URL；第 3 个参数-d'{...}'使用-d 标记来指示 CURL 发送跟随这个标记的 HTTP POST 数据。

可以用网页浏览器 Links 访问 Elasticsearch：

```
# links http://localhost:9200/
```

head 插件是 Elasticsearch 集群的 Web 前端，作为一个单独的 Web 应用而运行。在 Linux 系统下安装 head 插件的过程有如下几步。

（1）安装 npm。npm 是一个 Node 包管理和分发工具。安装命令如下：

```
# yum install npm
```

（2）下载 elasticsearch-head。可以用 git 命令克隆出一个小的本地仓库。

```
# git clone git://github.com/mobz/elasticsearch-head.git
# cd elasticsearch-head
```

（3）安装包。

```
# npm install
```

（4）启动 npm。

```
# npm run start
```

为了避免出现跨域问题，在文件 elasticsearch.yml 中添加如下配置：

```
http.cors.enabled: true
http.cors.allow-origin: "*"
```

可以发送 SIGTERM 信号停止服务，进程能捕捉到该信号并“干净地”关闭程序。首先找到进程的编号，然后使用 kill 命令通过指定 PID 给对应的进程发送 SIGTERM 信号。可以使用 JPS 命令找到 PID，然后执行如下命令：

```
# kill -15 PID
```

1.4.2　运行 Elasticsearch 作为服务进程

启动 Elasticsearch 的脚本放在/etc/init.d/目录下。启动脚本的写法如下：

```
case "$1" in
    start)
        do start-thing;
        ;;
    stop)
        do stop-thing;
        ;;
    restart)
        do restart-thing;
        ;;
    ...
esac
```

与 Windows 有两种启动模式“安全模式”和“正常启动”一样，Linux 一般有以下 7 个运行级别。

- 0：系统停机模式，系统默认运行级别不能设置为 0，否则不能正常启动，机器关闭。
- 1：单用户模式，root 权限，用于系统维护，禁止远程登录，类似 Windows 下的安全模式登录。
- 2：多用户模式，没有 NFS 网络支持。
- 3：完整的多用户文本模式，有 NFS，登录后进入控制台命令行模式。
- 4：系统未使用，保留，一般不用。
- 5：图形化模式，登录后进入图形 GUI 模式，X Window 系统。
- 6：重启模式，默认运行级别不能设为 6，否则不能正常启动。运行 init 6 机器就会重启。

查看运行级别，显示如下：

```
# runlevel
N 3
```

这里用 N 表示不存在上一次运行级别。

使用 chkconfig 命令可以设置开机时需要自动启动的服务程序。chkconfig 在没有参数运行时会列出所有的系统服务，等同于 chkconfig --list。

```
# chkconfig
aegis           0:off   1:off   2:on    3:on    4:on    5:on    6:off
agentwatch      0:off   1:off   2:on    3:on    4:on    5:on    6:off
jexec           0:off   1:on    2:on    3:on    4:on    5:on    6:off
netconsole      0:off   1:off   2:off   3:off   4:off   5:off   6:off
network         0:off   1:off   2:on    3:on    4:on    5:on    6:off
```

Elasticsearch 系统服务脚本主要内容如下：

```
#!/bin/bash
# chkconfig: 2345 10 90
# description: Elasticsearch Service ....
#定义一些需要用到的变量
ES_HOME=/opt/modules/elasticsearch-5.0.1
EXEC_PATH=$ES_HOME
EXEC=elasticsearch
DAEMON=$EXEC_PATH/bin/$EXEC
PID_FILE=$ES_HOME/pid/es.pid
```

```
ServiceName='Elasticsearch 5.0'

. /etc/rc.d/init.d/functions    #导入/etc/init.d/functions 中定义的方法
 #检查文件是否存在及是否可执行
if [ ! -x $DAEMON ] ; then
      echo "ERROR: $DAEMON not found"
      exit 1
fi
 #定义停止服务的方法
stop()
{
      echo "Stoping $ServiceName ..."
      ps aux | grep "$DAEMON" | kill -9 `awk '{print $2}'` >/dev/null 2>&1
      rm -f $PID_FILE
      usleep 100
      echo "Shutting down $ServiceName: [  successful  ]"
}
  #定义启动服务的方法
start()
{
      echo "Starting $ServiceName ..."
      $DAEMON > /dev/null &
      pidof $EXEC > $PID_FILE
      usleep 100
      echo "Starting $ServiceName: [  successful  ]"
}
```

将 Elasticsearch 服务开启并设置启动级别：

```
# chkconfig --level 3 es on
```

用 service（脚本文件名）start 来启动 Elasticsearch 服务：

```
# service es start
```

1.5　实现一个简单的网站搜索

首先在命令行检查 Elasticsearch 群集，然后通过 Java 客户端定义索引结构并导入数据，最后实现 JPS 搜索界面。这里通过 CURL 和 Elasticsearch 打交道，例如：

```
# curl http://localhost:9200/
```

输出结果如下：

```
{
  "name" : "yqrDfhV",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "EeaB7bHuTpGs1vy-CDy_uw",
  "version" : {
    "number" : "5.6.2",
    "build_hash" : "57e20f3",
    "build_date" : "2017-09-23T13:16:45.703Z",
    "build_snapshot" : false,
    "lucene_version" : "6.6.1"
```

```
  },
  "tagline" : "You Know, for Search"
}
```

可以把返回结果重定向到 less 命令，也就是把 CURL 的输出作为 less 命令的输入。

```
# curl -s http://localhost:9200/ | less
```

也可以重定向到文件，也就是把 CURL 的输出存储为文本文件。

```
# curl  -s  http://localhost:9200/ > es.txt
```

但是有些输出格式不可读。例如：

```
# curl http://localhost:9200/_cluster/health
```

所以采用 HTTPie 工具（下载网址是 https://httpie.org/）。HTTPie 工具可以输出方便阅读的 JSON 格式并在输出中加颜色。

pip 是 Python 的包管理工具，可以使用 pip 安装 HTTPie 工具：

```
# pip install --upgrade httpie
```

如果由于某种原因 pip 安装失败，可以尝试使用命令 easy_install httpie 作为安装 HTTPie 的后备方法。

在做任何事情之前，首先需要了解 Elasticsearch 群集是否健康。有几种方法可以收集这些信息，但最容易且最方便的是使用 Cluster API，特别是集群健康端点。运行如下命令：

```
$ http http://localhost:9200/_cluster/health
```

输出结果如下：

```
HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
    "active_primary_shards": 0,
    "active_shards": 0,
    "active_shards_percent_as_number": 100.0,
    "cluster_name": "es-catalog",
    "delayed_unassigned_shards": 0,
    "initializing_shards": 0,
    "number_of_data_nodes": 3,
    "number_of_in_flight_fetch": 0,
    "number_of_nodes": 3,
    "number_of_pending_tasks": 0,
    "relocating_shards": 0,
    "status": "green",
    "task_max_waiting_in_queue_millis": 0,
    "timed_out": false,
    "unassigned_shards": 0
}
```

在这些细节中，寻找应该设置为绿色的状态指示器，这意味着所有分片都被分配，并且集群处于良好的运行状态。

结果显示我们的 Elasticsearch 集群都是绿色的，准备好后就可以去干活了。下一个逻

辑步骤是创建一个目录索引，但是在此之前让我们先检查一下，使用 Indices API 是否已经创建了任何索引。

```
$ http http://localhost:9200/_stats
```

输出结果如下：

```
HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
    "_all": {
        "primaries": {},
        "total": {}
    },
    "_shards": {
        "failed": 0,
        "successful": 0,
        "total": 0
    },
    "indices": {}
}
```

正如预期的那样，我们的集群还没有任何内容。

1.5.1　定义索引结构

首先在 Eclipse 中创建一个 Maven 项目，增加对 Easticsearch 和 Transport 的引用。pom.xml 文件部分内容如下：

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>2.8.1</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.8.1</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-annotations</artifactId>
        <version>2.8.1</version>
    </dependency>
```

```
    <dependency>
        <groupId>org.elasticsearch.client</groupId>
        <artifactId>transport</artifactId>
        <version>5.6.2</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.elasticsearch/
    elasticsearch -->
    <dependency>
        <groupId>org.elasticsearch</groupId>
        <artifactId>elasticsearch</artifactId>
        <version>5.6.2</version>
    </dependency>

</dependencies>
```

为了发送查询请求，首先需要和 Easticsearch 集群建立连接。测试连接代码如下：

```
public class TestClient {
static TransportClient client;
static String clusterName = "elasticsearch";
static String serverhost = "localhost";          //IP 地址
static int serverPort = 9300;                    //端口号

public static void main(String[] args) throws UnknownHostException {
    Settings settings = Settings.builder().put("cluster.name",
    clusterName)
            .build();
    client = new PreBuiltTransportClient(settings)
            .addTransportAddress(new InetSocketTransportAddress
            (InetAddress
                    .getByName(serverhost), serverPort));

    System.out.println("######创建了 ElasticClient ##############
    ####");
    System.out.println("--------------------");
    client.close();
}
}
```

在命令行运行这个测试类，检查能否和 Easticsearch 集群成功建立连接：

```
# mvn compile exec:java -Dexec.mainClass=demo_elastic.demo_elastic.
TestClient
```

首先定义索引库结构：

```
private static XContentBuilder getMapping(String indexType)
    throws Exception {
XContentBuilder mapping = XContentFactory.jsonBuilder().startObject()
        .startObject(indexType).startObject("properties")

        //定义标题列
        .startObject("title").field("type", "string")
        .field("store", "yes").field("analyzer", "standard")
        .endObject()
```

```
        //定义内容列
        .startObject("body").field("type", "string")
        .field("store", "yes").field("analyzer", "standard")
        .endObject()

        .endObject()                                    // 属性结束
        .endObject()                                    // 索引类型结束
        .endObject();
return mapping;
}
```

然后调用 IndicesAdminClient.putMapping()方法设定索引库结构：

```
String indexName = "cms";

IndicesAdminClient ac = client.admin().indices();
CreateIndexRequestBuilder builder = ac.prepareCreate(indexName);

//设置分片数量
Builder setting = Settings.builder().put("number_of_shards", 1);
builder.setSettings(setting);

// 首先创建索引库
CreateIndexResponse indexresponse = client.admin().indices()
    // 这个索引库的名称不能包含大写字母
    .prepareCreate(indexName).setSettings(setting.build()).execute()
    .actionGet();
System.out.println("CreateIndex "+indexresponse.isAcknowledged());
                                                    //看是否成功创建索引
//然后设定索引库结构
String type = "article";
XContentBuilder mapping = getMapping(type);
PutMappingRequest mappingRequest = Requests
    .putMappingRequest(indexName).type(type).source(mapping);
PutMappingResponse putMappingResponse = client.admin().indices()
    .putMapping(mappingRequest).actionGet();
//看是否成功设定索引结构
System.out.println("putMappingResponse " + putMappingResponse.
isAcknowledged());
```

检查索引结构是否已经成功设定：

```
# http http://localhost:9200/cms/_settings
```

输出结果如下：

```
HTTP/1.1 200 OK
content-encoding: gzip
content-length: 176
content-type: application/json; charset=UTF-8

{
   "cms": {
      "settings": {
         "index": {
```

```
                "creation_date": "1508639667017",
                "number_of_replicas": "1",
                "number_of_shards": "1",
                "provided_name": "cms",
                "uuid": "u7NGm_aOQQGp7zSgWERUOA",
                "version": {
                    "created": "5060299"
                }
            }
        }
    }
}
```

使用 IndicesAdminClient.prepareDelete()方法删除索引。

```
IndicesAdminClient admin = client.admin().indices();
admin.prepareDelete(indexName).execute().actionGet().isAcknowledged();
```

1.5.2 导入数据

使用 IndexRequestBuilder 插入数据。首先通过 Client.prepareIndex()方法指定索引名称、类型名称和文档的唯一编号，然后调用 IndexRequestBuilder. setSource()方法设定文档内容，最后调用 IndexRequestBuilder.execute()方法并返回结果。示例代码如下：

```
String id="20";                                          //唯一列的值
IndexRequestBuilder indexRequestBuilder = client.prepareIndex(
        "cms", "article", id);
//准备文档内容
Map<String, String> source = new HashMap<>();
source.put("title", "标题");
source.put("body", "内容");
indexRequestBuilder.setSource(source);
IndexResponse response = indexRequestBuilder.execute().actionGet();
System.out.println(response.status().name()); // 如果成功，则返回 CREATED
```

命令行检查索引库：

```
# curl -i http://localhost:9200/cms/article/_search?pretty -d '
{
    "size": 10,
    "query": {
        "match_all" : {
        }
    }
}'
```

Client.prepareDelete()方法删除数据：

```
String id="20";                                          //唯一列的值

DeleteResponse reponse = client.prepareDelete("cms","article",id).get();
System.out.println(reponse.status().name());   //成功就返回 OK
```

使用 UpdateRequest 对象更新数据：

```
String id = "20";                                        // 唯一列的值
```

```
UpdateRequest updateRequest = new UpdateRequest();
updateRequest.index("cms");                    // 索引名
updateRequest.type("article");                 // 类型
updateRequest.id(id);                          // ID
updateRequest.doc(XContentFactory.jsonBuilder().startObject()
        .field("body", "content").endObject());
UpdateResponse resp = client.update(updateRequest).get();
System.out.println(resp.getResult().name());       //如成功就返回 UPDATED
```

1.5.3　查询 API

我们构造一个 QueryBuilder 对象来查询文档：

```
QueryBuilder qb = QueryBuilders.matchAllQuery();
System.out.println(qb);
```

输出结果如下：

```
{
  "match_all" : {
    "boost" : 1.0
  }
}
```

完整的查询代码如下：

```
MatchAllQueryBuilder qb = QueryBuilders.matchAllQuery();
String index = "cms";
SearchResponse searchResponse =

client.prepareSearch(index).setQuery(qb).execute().actionGet();

SearchHits hits = searchResponse.getHits();
for (SearchHit hit : hits) {
System.out.println("id "+hit.getId());     // 文档 ID
Map<String, Object> result = hit.getSource();
                                           // 键是列名，值是文档中该列的值
for (final Entry<String, Object> entry : result.entrySet()) {
    System.out.println(entry.getKey() + " : " + entry.getValue());
                                           //输出文档内容
}
}
```

基本的关键词查询：

```
String keyWords = "DNA";                   //查询词
QueryStringQueryBuilder qb = new QueryStringQueryBuilder(keyWords);
String index = "cms";                      //索引名

SearchResponse searchResponse = client.prepareSearch(index)
    .setQuery(qb).execute().actionGet();

//遍历查询结果
SearchHits hits = searchResponse.getHits();
for (SearchHit hit : hits) {
```

```
System.out.println("id " + hit.getId());   // 文档 ID
Map<String, Object> result = hit.getSource();
                                          // 键是列名，值是文档中该列的值
for (final Entry<String, Object> entry : result.entrySet()) {
    System.out.println(entry.getKey() + " : " + entry.getValue());
}
}
```

SearchRequestBuilder.setFetchSource()方法限定返回的列数据，以避免返回过多的数据：

```
SearchRequestBuilder restBuilder = client.prepareSearch(index).
setQuery(qb);
//限定只返回 title 列的数据
SearchResponse searchResponse =restBuilder.setFetchSource("title", null)
        .execute().actionGet();
```

也可以调用 Client.prepareGet()方法只返回指定文档，代码如下：

```
GetResponse response = client.prepareGet("cms", "artile", "20")
    .setFetchSource("title",null)
    .execute().actionGet();
Map<String, Object> result = response.getSourceAsMap();   //结果封装成 Map
for (final Entry<String, Object> entry : result.entrySet()) {
System.out.println(entry.getKey() + " : " + entry.getValue());
}
```

为了实现分页显示，需要设置两个参数：从第几个结果开始返回文档，以及最多返回多少个文档。通过 SearchRequestBuilder 的 setFrom()和 setSize()方法来设置这两个参数。参考代码如下：

```
int rows=10;                                          //一页显示多少条搜索结果
int offset=0;                                         //开始行
SearchRequestBuilder searchRequestBuilder = client.prepareSearch(index);
// 分页应用
searchRequestBuilder.setFrom(offset).setSize(rows);
```

另外，记录结果总数，用于计算总的页码数量。

```
SearchHits searchHits = response.getHits();
long totalHits = searchHits.getTotalHits();     //得到结果总数
```

为了实现查询结果高亮显示，首先指定高亮标签及哪些列需要高亮，然后在符合条件的结果中得到高亮段。通过 HighlightBuilder 指定高亮相关的信息。

```
SearchRequestBuilder searchRequestBuilder = client.prepareSearch(index);
// 高亮标签
HighlightBuilder hiBuilder = new HighlightBuilder();
hiBuilder.preTags("<span style=\"color:red\">");
hiBuilder.postTags("</span>");
// 指定高亮字段
hiBuilder.field("title");

searchRequestBuilder.highlighter(hiBuilder);
```

1.5.4　实现搜索界面

本节将使用成熟的 JSP 技术实现搜索页面。首先介绍 Web 服务器 Tomcat 的下载、安装和使用过程，然后介绍使用 Taglib 开发搜索界面。

在 PC 端或者手机端可以使用的搜索界面由 Web 服务器提供，Web 服务器提供了 Servlet 开发接口让应用程序生成返回给浏览器的数据。Servlet 类运行在 Servlet 容器中，Catalina 是 Web 服务器 Tomcat 的 Servlet 容器，可以从 http://tomcat.apache.org/网站上找到 Tomcat 的具体下载地址。

通过 SSH 客户端登录 Linux 服务器，然后用 wget 命令下载 Tomcat 压缩包。

```
wget
http://mirrors.hust.edu.cn/apache/tomcat/tomcat-8/v8.5.23/bin/apache-
tomcat-8.5.23.tar.gz
```

下载之后解压缩：

```
tar -xvf apache-tomcat-8.5.23.tar.gz
```

为了简单测试 Servlet 类，可以直接重写 Tomcat 自带的 Servlet 类似例子，如 HelloWorld Example，然后增加 Tomcat 所使用的内存。修改配置文件 catalina.sh 如下：

```
vi /usr/local/apache-tomcat-8.5.23/bin/catalina.sh
```

在文件 catalina.sh 的开始位置增加如下行：

```
JAVA_OPTS=-Xmx1024m
```

修改 Tomcat 配置文件 server.xml，把监听端口号从 8080 改到 80，并且支持 UTF-8 编码：

```
vi /usr/local/ apache-tomcat-8.5.23/conf/server.xml
```

在 server.xml 配置文件中增加如下配置：

```
useBodyEncodingForURI="true" URIEncoding="UTF-8"
```

可以把 Web 应用打一个 war 包，然后传到服务器上的 webapps/子路径下，这样会自动解压缩 WAR 包中的 Web 应用。

浏览器用户在输入框中输入查询词，查询词一般会作为 GET 请求中的参数提供给 Web 服务器。然后通过 GET 请求将查询词提交到 Search.jsp 的 HTML 网页，代码如下：

```
<form method="get" action="./Search.jsp">
<input type="text"/>
<input type=submit value=搜索>
</form>
```

在 HTML 5 中引入搜索输入框代替文本输入框：

```
<input type=search>
```

为了实现代码复用，再定义一个专用于根据关键词查询返回结果的 Taglib。搜索结果页是一个表格型的数据。Listlib 实现了对数据的封装和抽象，可以通过它来控制显示的结果数量，如可以指定每页显示 20 条或 10 条记录。执行 Lucene 搜索的类继承 List

Creator 接口，并把搜索结果通过 ListContainer 类的实例返回即可。Listlib 中定义的标签有以下几类：

1．init标签

init 是 Listlib 中的起始标签。其创建一个 ListCreator 对象，并且运行该对象的 execute() 方法，同时把它存储在 HttpServletRequest 属性中。这是一个容器标签，所以在 JSP 页面使用时，其他的 TAG 都必须嵌套在这个 TAG 中间。

init 标签的主要属性有：通过 name 指定一个名字，因为需要通过该名字把 ListCreator 对象存储在 HttpServletRequest 属性中；通过 listCreator 来指定创建 ListCreator 的对象；通过 max 声明每页必须显示的记录条数。

2．hasResults标签

如果用户查询词有匹配的文档，则会执行 hasResults 标签，否则会跳过。

3．hasNoResults标签

如果用户查询词没有匹配的文档则会执行 hasNoResults 标签，否则会跳过。

4．prop标签

返回 list 中的属性值。和搜索结果总体相关的信息可以通过 prop 标签来显示，如搜索提示词、搜索结果分类统计等。

5．hasPrev标签

如果还可以继续往回遍历，则会显示 hasPrev 标签中的内容，否则就跳过。

6．hasNext标签

如果还可以继续向下遍历，则会显示 hasNext 标签中的内容，否则就跳过。

7．iterate标签

遍历 ListContainer 中的元素。

8．iterateProp标签

从迭代器的当前对象返回其中的属性值，如返回标题，通过命令<list:iterateProp property="title"/>，可以在 listlib.tld 文件中定义 Taglib。

以中文字符作为参数时，需要用对应字符集编码这个字符串。这里为 URL 编码专门写了一个自定义标签 iteratePropURLEncodeTag。

使用 iterateURLEncodeProp 的例子：

```
<a href=
"folder.jsp?folder=<list:iterateURLEncodeProp
property="folder"/>&docType=<list:iterateProp property="docType"/>"
执行搜索的 Java Bean:
public class SearchWeb implements ListCreator {
private String _query;

private Client server;                                    //在 Web 容器内全局唯一
private static Logger logger = Logger.getLogger(SearchWeb.class.
getName());

//只调用一次
public void init(String host) throws Exception {
    server = new PreBuiltTransportClient(Settings.EMPTY)
    .addTransportAddress(new InetSocketTransportAddress(InetAddress
            .getByName(host), 9300));
}
}
```

在开发搜索 Web 界面之前，我们先写一个控制台方式运行的搜索程序测试一下索引库。使用存根类专门测试 ListCreator 的实现类 SearchWeb：

```
String host = "localhost";                                //Elasticsearch 服务地址
String query = "DNA";                                     //查询词
SearchWeb search = new SearchWeb();
search.init(host);
search.setQuery(query);
//翻页参数
int offset = 0;
int max = 10;
PageContextImpl context = new PageContextImpl();          //PageContext 存根类
context.request = new ServletRequestImpl();               //ServletRequest 存根类
ListContainer lc = search.execute(context, offset, max);
lc.setUrl("Search.jsp");
System.out.println("result size:" + lc.getSize());//输出结果总数
Iterator it = lc.getIterator();
while (it.hasNext()) {
HashMap<String, String> row = (HashMap<String, String>) it.next();
System.out.println("url:" + row.get("url"));          //输出网址列
System.out.println("title:" + row.get("title"));      //输出标题列
System.out.println("body:" + row.get("body"));        //输出内容列
}
```

在 JSP 调用实现类之前，需要区分哪些类是静态的、全局唯一的，哪些对象需要在页面内即时创建和使用，哪些对象在整个用户会话期间内有效。

使用 jsp:useBean 标签创建一个 Bean 实例并指定它的名字和作用范围。

```
<!--定义全局唯一的搜索 Bean - 它实现了 ListCreator 的 execute 方法 -->
<jsp:useBean id="searchInf" class="com.lietu.search.SearchWeb" scope="
application">
<!--指定 Elasticsearch 服务器的 IP 地址-->
<% searchInf.init("localhost"); %>
</jsp:useBean>
```

```
<!--用从 HTTP 的 get 参数得到的查询中设置搜索对象的属性-->
<jsp:setProperty name="searchInf" property="query" value="<%=query%>"/>
<!--执行搜索并把返回结果封装到 ListContainer -->
<list:init name="information"  listCreator="searchInf" max="20">
```

搜索结果翻页使用 JSP 标签库 Pager-taglib。使用 Pager-taglib 的流程如下：

（1）复制 pager-taglib.jar 包到 lib 目录下，不需要改 web.xml。

（2）在 JSP 页面中使用 taglib 指令引入 pager-taglib 标签库。

（3）使用 pager-taglib 标签库进行分页处理。

通过 maxPageItems 参数设定每页最多显示的结果数。在 JSP 页面中使用翻页标签库的例子如下：

```
<pg:pager   url="Search.jsp"
    items="<%=Integer.parseInt(listSize)%>"
    maxPageItems="20"
    maxIndexPages="10"
    export="currentPageNumber=pageNumber"
    scope="request">
 <pg:param name="query" value="<%=query%>"/>
…
</pg:pager>
```

其中，在 pg:pager 标签中定义了 action 的 URL 地址，在 pg:param 标签中定义了查询参数 query。

pager-taglib 在输出的页面中生成链接 search.jsp?query=%E7%9A%84&pager.offset=10，其中包含了开始位置的参数。在 InitTag 类中得到开始返回结果的偏移量。

```
public static final String OFFSET_KEY = "pager.offset";

public int doStartTag() throws JspException {
//得到字符串形式的参数
String offsetStr = pageContext.getRequest().getParameter(OFFSET_KEY);
//转换成整数
int offset = Integer.parseInt(offsetStr);
}
```

头部代码如下：

```
1  <%@ page contentType="text/html; charset=UTF-8"%>
2  <%@ page session="false" %>
3  <%@ page import="java.net.URLEncoder"%>
4  <%@ taglib uri="http://jsptags.com/tags/navigation/pager" prefix="pg" %>
5  <%@ taglib uri="http://com.bitmechanic/listlib" prefix="list" %>
6  <%
7    String query = request.getParameter("query");
8    if (query == null) query = "";
9  %>
10  <html>
11  <head>
12     <title>全文检索</title>
13  </head>
14
15  </html>
```

其中，第 1～5 行是一些 JSP 指令，第 2 行表示不使用 session 来保存信息，第 3 行用于引入包，第 6～9 行是 JSP 中的 Java 代码。

内容部分的代码如下：

```
1   <body>
2  <table width="998" border="0" align="center" cellpadding="0" cellspacing=
     "0">
3     <tr>
4      <td colspan="4"><% if (!query.equals("")) { %>
5         <jsp:useBean id="searchInf" class="com.lietu.search.SearchWeb"
           scope="application">
6       <% searchInf.init("localhost"); %>
7       </jsp:useBean>
8
9        <jsp:setProperty name="searchInf" property="query" value="
          <%=query%>"/>
10       </td>
11    </tr>
12    <tr>
13      <td valign="top" width="80%">
14  <%long start = System.currentTimeMillis();%>
15  <list:init name="information"  listCreator="searchInf" max="20">
16  <% long end = System.currentTimeMillis();%>
17  <list:hasResults>
18  <pg:pager       url="./Search.jsp"
19      items="<%=Integer.parseInt(listSize)%>"
20      maxPageItems="20"
21      maxIndexPages="10"
22      export="currentPageNumber=pageNumber"
23      scope="request"> <pg:param name="query" value="<%=query%>"/>
24  <pg:page export="firstItem, lastItem">
25       <div class="resultInfo">
26    <TABLE cellSpacing=0 cellPadding=0 width="98%" border=0>
27         <TBODY>
28         <TR>
29           <TD bgColor=#ffffcc height=30>    
             有<strong><%=listSize%></strong>项符合<strong><%=query%>
             </strong>的查询结果，以下是第 <strong><%= firstItem %>-<%=
             lastItem %></strong> 项。（搜索用时<%=((double)end-(double)
             start)/1000%>秒）</TD>
30         </TR></TBODY></TABLE>
31       </div>
32       </pg:page>
33  <TABLE width="95%" border="0" align="center" cellpadding="2"
    cellspacing="0">
34  <list:iterate>
35      <TR>
36        <td width=66%><a href="<list:iterateProp property="url"/>"
          target="_blank" ><B><FONT style="FONT-SIZE: 14px"><list:
          iterateProp property="title"/></FONT></B>  <FONT
          size=-1 color=#6f6f6f></FONT></td>
37        <td width=18%></td>
38        <td width=16%></td>
39      </TR>
```


```
40      <TR vAlign=top >
41        <td colspan=3> <FONT size=-1><list:iterateProp property="
          body"/></FONT>
42        </td>
43      </TR>
44      <TR vAlign=top  >
45        <td colspan=3> <span class=tailurl><a class=tail href="<list:
          iterateProp property="url"/>" target="_blank" ><list:
          iterateProp property="url"/></a></span>
46        </td>
47      </TR>
48    </list:iterate>
49    </TABLE>
50       <pg:index export="totalItems=itemCount">
51       <div class="rnav"> <span class="rnavLabel">   
              结果: </span>  <pg:prev
         export="pageUrl">
52         <a href="<%= pageUrl %>"class="rnavLink">&#171; 
          上一页</a> 
53         </pg:prev> <pg:pages>
54         <%
55       if (pageNumber.intValue() < 10) {
56         %>
57          
58         <%
59       }
60       if (pageNumber == currentPageNumber) {
61         %>
62         <b><%= pageNumber %></b>
63         <%
64       } else {
65         %>
66         <a href="<%= pageUrl %>"><%= pageNumber %></a>
67         <%
68       }
69     %>
70         </pg:pages> <pg:next export="pageUrl">   <a href="<%=
           pageUrl %>" class="rnavLink">下一页 &#187;</a>
71         </pg:next> </div>
72       </pg:index> </pg:pager> </list:hasResults> <list:hasNoResults>
             对不起，没有结果返回。
73       <p>          </list:hasNoResults>
74  </list:init>
75     <br>
76     <% } else{ %>
77  <br>
78       <% }%><br>
79     </td>
80   </tr>
81   <tr>
82     <td align="center" bgcolor="#EEEEEE"></td>
83   </tr>
84  </table>
85  </body>
```


其中，第 5～7 行中定义了搜索用的 JavaBean，第 9 行设置查询词，第 14～16 行记录搜索用时，第 18～24 行使用翻页标签库，第 29 行显示搜索结果用时，第 34～48 行显示搜索结果，第 50～72 行显示翻页相关信息，第 72 行和第 73 行处理无返回结果的情况。

1.6　本章小结

本章从搜索引擎的需求出发，使用 Elasticsearch 的 Java API 实现了一个简单的搜索界面，可以通过 Mapping 定义索引结构，然后填充数据到索引，最后再做搜索部分。除了使用 Java API，还可以使用.NET、Python 和 PHP 开发 Elasticsearch 搜索客户端。

Elasticsearch 对外使用 JSON 和客户端实现数据交换。JSON 是一种与语言无关的数据格式。

CURL 是 Linux 下的一个 HTTP 命令行工具，通过 CURL 发送命令和 Elasticsearch 节点交互。

除了自己写 Shell 脚本外，还可以使用 Java Service Wrapper 将 Elasticsearch 部署成服务。

Elasticsearch 的引擎是一个运行时环境，使人们能够实时地分析和处理大量数据，它“不要求你必须是一位数据科学家才能把它用好”，这也是该产品的一个主要亮点。Elasticsearch 存在的理由是把大数据的复杂性简单化。

从 Elasticsearch5.x 版开始，它的服务器端代码开始成熟，但客户端代码仍然在发展和调整之中。Java API 与 Elasticsearch 服务器在端口 9300 上打交道，而 RESTful 的 HTTP 客户端 Jest 使用端口 9200。Jest 提供自己的 Java API，还可以使用 Elasticsearch Java API 来构建查询，然后提交给 RESTful 端点。从 5.0 版开始，Elasticsearch 推出了自己的 Rest Client。

除了 Elasticsearch，还可以使用 Solr 实现网站搜索。

第 2 章　开发中文搜索引擎

Elasticsearch 中的全文索引是按词组织的。词是怎么来的呢？对于中文文档来说，是中文分词分出来的。例如，如果没有中文分词，搜索“印度”，则会出现“印度尼西亚”相关信息。

本章首先介绍中文分词的基本原理，然后介绍开发支持 Lucene 的中文分词，以及开发 Elasticsearch 所需要的 AnalyzerProvider，最后介绍通过增加 *n* 元连接来提高分词准确度的方法。

2.1　中文分词原理

中文分词目前常用的方法有固定规则的最长匹配方法和基于机器学习的概率语言模型方法，下面具体介绍。

2.1.1　最长匹配方法

假如要切分“印度尼西亚潜水”这句话，希望切分出“印度尼西亚”，而不希望切分出“印度”这个词，找最长词是最大长度匹配的思想。写作者（即被分词代码分析文本的写作者）倾向于使用更短的词，除非有必要才用长词表述。

最大长度匹配的分词方法实现起来很简单。每次从词典中寻找和待匹配串前缀最长匹配的词，如果找到匹配词，则把这个词作为切分词，待匹配串减去该词；如果词典中没有词能匹配，则按单字切分。例如，Trie 树结构的词典中包括如下 5 个词语：

印　印度　印度尼西亚　潜水　水

输入“印度尼西亚潜水”，首先匹配出开头的最长词“印度尼西亚”，然后匹配出“潜水”。切分过程如图 2-1 所示。

最后分词结果为“印度尼西亚/潜水”。

在分词类 Segmenter 的构造方法中输入要处理的文本，然后通过 nextWord()方法遍历单词，text 变量记录切分文本，offset 变量记录已经切分到哪里。分词类基本实现如下：

```
public class Segmenter {
    String text = null;                    //切分文本
```

```
    int offset;                                    //已经处理到的位置

    public Segmenter(String text) {
        this.text = text;                          //更新待切分的文本
        offset = 0;                                //重置已经处理到的位置
    }

    public String nextWord() {                     //得到下一个词，如果没有则返回 null
        // 返回最长匹配词，如果没有匹配上，则按单字切分
    }
}
```

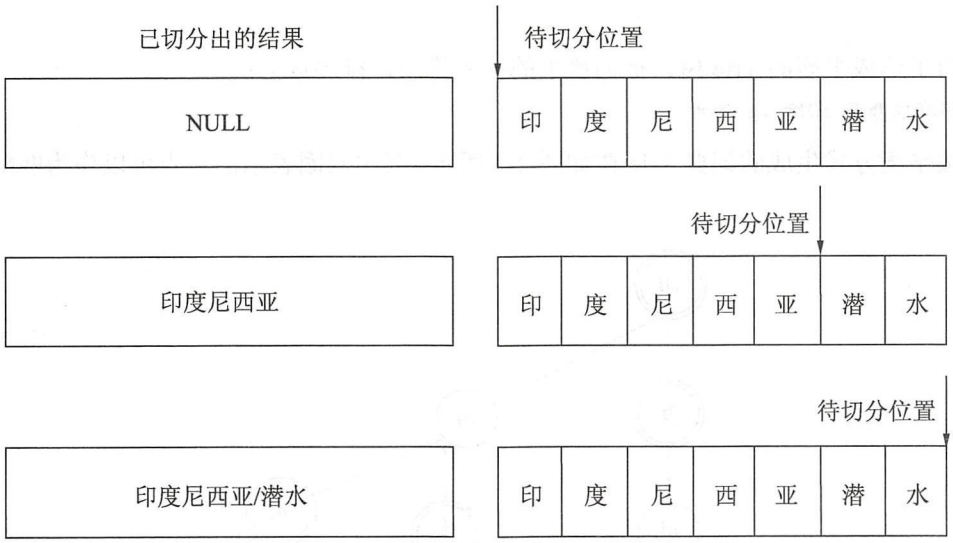


图 2-1　最大长度匹配切分过程

为了避免重复加载词典，在 Segmenter 类的静态方法中加载词典。代码如下：

```
private static TSTNode root;                       //根节点是静态的

static {                                           //加载词典
    String fileName = "WordList.txt";              //词典文件名
    try {
        FileReader fileRead = new FileReader(fileName);
        BufferedReader read = new BufferedReader(fileRead);
        String line;                               //读入的一行
        try {
            while ((line = read.readLine()) != null) {     //按行读
                StringTokenizer st = new StringTokenizer(line, "\t");
                String key = st.nextToken();       //得到词
                TSTNode endNode = createNode(key);
                                                   //创建词对应的结束节点并返回
                //设置这个节点对应的值，也就是把它标记成可以结束的节点
                endNode.nodeValue = key;
```


```
                }
            } catch (IOException e) {
                e.printStackTrace();
            }finally {
                read.close();                          //关闭读入流
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

为了形成平衡的 Trie 树，把词典中的词先排序，排序后为：

印度尼西亚 印度 印 潜水 水

按平衡方式生成的词典 Trie 树如图 2-2 所示，其中双圈表示的节点可以作为匹配终止节点。

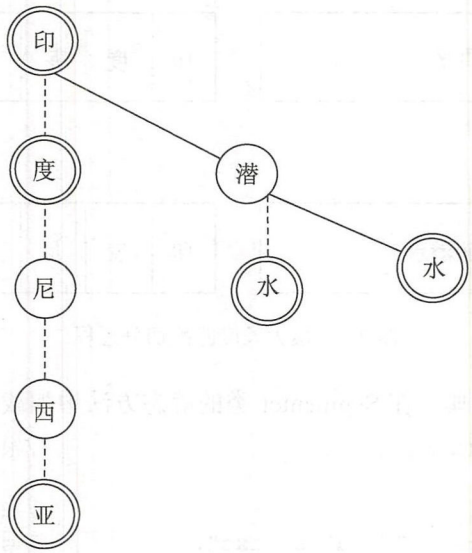


图 2-2　三叉树

在最大长度匹配的分词方法中，需要用到从待切分字符串返回从指定位置（offset）开始的最长匹配词的方法。例如，当输入串是“印度尼西亚潜水”，则返回“印度尼西亚”这个词，而不是返回“印”或者“印度”，匹配的过程就像一条蛇爬上一棵树一样。例如，当 offset=0 时，找最长匹配词的过程如图 2-3 所示。树上有一个当前节点，输入字符串中有一个当前位置。图中用数字标出了匹配过程中第一步、第二步和第三步中当前节点和当前位置分别位于什么位置。

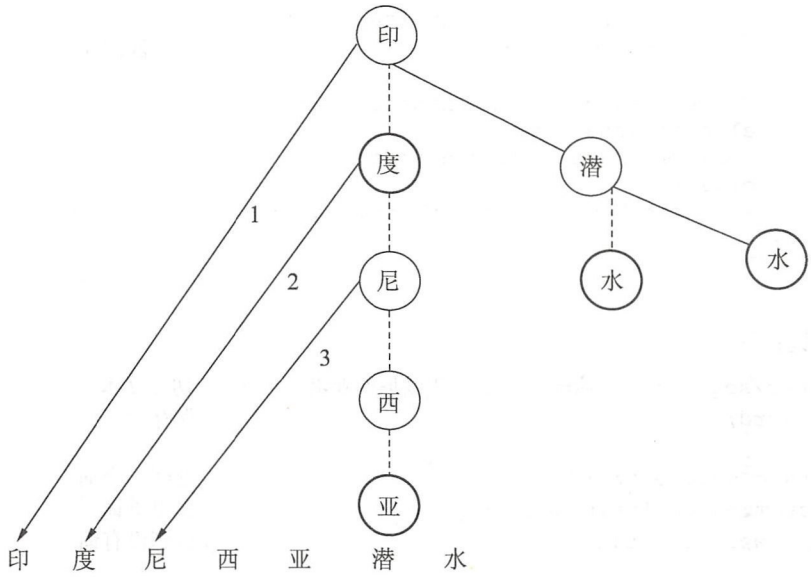


图 2-3　找最长匹配词

从 Trie 树搜索最长匹配单词的方法如下：

```
public String nextWord() {                          //得到下一个词
    String word = null;                             //候选最长词
    if (text == null || root == null) {
        return word;
    }
    if (offset >= text.length())                    //已经处理完毕
        return word;
    TSTNode currentNode = root;                     //从根节点开始
    int charIndex = offset;                         //待切分字符串的处理开始位置
    while (true) {
        if (currentNode == null) {                  //已经匹配完毕
            if(word==null){                         //没有匹配上，则按单字切分
                word = text.substring(offset,offset+1);
                offset++;
            }
            return word;                            //返回找到的词
        }
        int charComp = text.charAt(charIndex) - currentNode.splitChar;
                                                    //比较两个字符

        if (charComp == 0) {
            charIndex++;                            //找字符串中的下一个字符

            if (currentNode.nodeValue != null) {
                word = currentNode.nodeValue;       // 候选最长匹配词
                offset = charIndex;                 //设置偏移量
            }
```

```
            if (charIndex == text.length()) {
                return word;                          //已经匹配完
            }
            currentNode = currentNode.mid;
        } else if (charComp < 0) {
            currentNode = currentNode.left;
        } else {
            currentNode = currentNode.right;
        }
    }
}
```

测试分词：

```
Segmenter seg = new Segmenter("印度尼西亚潜水"); //切分文本
String word;                                    //保存词
do {
    word = seg.nextWord();                      //返回一个词
    System.out.println(word);                   //输出单词
} while (word != null);                         //直到没有词
```

返回结果如下：

```
印度尼西亚
潜水
null
```

可以给定一个字符串，然后枚举出所有的匹配点。以“大学生活动中心”为例，第一次调用时，offset 是 0，第二次调用时，offset 是 3。因为采用了 Trie 树结构查找单词，所以和用 HashMap 查找单词的方式比较起来这种实现方法的代码更简单，而且切分速度更快。

词典工厂类代码如下：

```
public interface DicFactory {
    TernarySearchTrie create();
}
```

实现从数据库表生成词典的类：

```
public class DicDBFactory implements DicFactory {

    public static Connection getConnect() {
        try {
            Class.forName("org.sqlite.JDBC");  //加载驱动程序
            String absolute_path_to_sqlite_db = "./dic/words.sqlite";
                                               //SQLIte 数据库文件
            //建立连接
            return DriverManager.getConnection("jdbc:sqlite:"+absolute_
            path_to_sqlite_db);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
```

```
    @Override
    public TernarySearchTrie create() {
        TernarySearchTrie dic = new TernarySearchTrie();

        Connection conn = getConnect();                    //得到数据库连接

        String sql = "SELECT WORD,FRQ from WORDS";         //查询词典表
        try {
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sql);         //执行 SQL 语句

            while (rs.next()) {
                String word = rs.getString(1);             //得到词
                dic.addWord(word);                         //向词典中增加词
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            conn.close();                                  //关闭数据库连接
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return dic;
    }

}
```

实现从文件中生成词典的类：

```
public class DicFileFactory implements DicFactory {
    public static final String binDic = "Words.bin";   //词典文件

    @Override
    public TernarySearchTrie create() {
        TernarySearchTrie dic = new TernarySearchTrie();
        File binFile = new File(binDic);

        if (!binFile.exists()) {                           //词典文件还不存在
            dic = (new DicDBFactory()).create();
            //生成二进制格式的词典文件
            dic.compileDic(binFile);
        } else {//从生成的数组树中进行加载
            //加载二进制格式的词典文件
            dic.loadBinaryDataFile(binFile);
        }

        return dic;
    }

}
```

2.1.2 自己写分析器

开发分词，除了要在分词项目中导入核心包 lucene-core-6.3.0.jar 之外，还要导入 lucene-analyzers-common-6.3.0.jar 包。

分词做好以后，要套上 Tokenizer 接口才能在 Lucene 中使用，最后用自定义的 Analyzer（分析器）调用支持中文的 Tokenizer 接口。

首先初始化 CharTermAttribute 和 OffsetAttribute 这样的属性：

```
public final class CnTokenizer extends Tokenizer {
    private final CharTermAttribute termAtt = addAttribute
    (CharTermAttribute.class);              //词
    private final OffsetAttribute offsetAtt = addAttribute
    (OffsetAttribute.class);                //位置
}
```

CnTokenizer()构造方法如下：

```
CnTokenizer(factory,input, dict);
```

下面是采用最大长度匹配实现的一个简单的 Tokenizer 接口。

```
public class CnTokenizer extends Tokenizer {
    private static TernarySearchTrie dic = new TernarySearchTrie
    ("SDIC.txt");                          //词典
    private CharTermAttribute termAtt; //词属性
    private static final int IO_BUFFER_SIZE = 4096;
    private char[] ioBuffer = new char[IO_BUFFER_SIZE];

    private boolean done;
    private int i = 0;                     // i 是用来控制匹配的起始位置的变量
    private int upto = 0;

    public CnTokenizer(Reader reader) {
        super(reader);
        this.termAtt = addAttribute(CharTermAttribute.class);
        this.done = false;
    }

    public void resizeIOBuffer(int newSize) {
        if (ioBuffer.length < newSize) {
            //不够大。创建一个新的数组，轻微地多分配并保留内容
            final char[] newCharBuffer = new char[newSize];
            System.arraycopy(ioBuffer, 0, newCharBuffer, 0, ioBuffer.
            length);
            ioBuffer = newCharBuffer;
        }
    }

    @Override
    public boolean incrementToken() throws IOException {
        if (!done) {
            clearAttributes();
```

```
                done = true;
                upto = 0;
                i = 0;
                while (true) {
                    final int length = input.read(ioBuffer, upto, ioBuffer.
                    length
                            - upto);
                    if (length == -1)
                        break;
                    upto += length;
                    if (upto == ioBuffer.length)
                        resizeIOBuffer(upto * 2);
                }
            }

            if (i < upto) {
                char[] word = dic.matchLong(ioBuffer, i, upto);
                                                    //最大长度匹配
                if (word != null) {                 //已经匹配上
                    termAtt.setTermBuffer(word, 0, word.length);
                    i += word.length;
                } else {
                    termAtt.setTermBuffer(ioBuffer, i, 1);
                    ++i;                            //下次匹配点在这个字符之后
                }
                return true;
            }
            return false;
        }
    }
```

为了能在 Lucene 中连续执行，CnTokenizer()构造方法中需要重写 reset()方法。

```
@Override
public void reset() throws IOException {
  super.reset();
  this.i=0;
  this.done = false;
  this.upto = 0;
}

@Override
public void reset(Reader input) throws IOException {
  super.reset(input);
  reset();
}

@Override
public void end() throws IOException {
  // 设置最后的偏移量
  final int finalOffset = correctOffset(upto);
  offsetAtt.setOffset(finalOffset, finalOffset);
}
```

中文中的“的”“了”等虚词在文档中出现频率较高，但一般不搜索这些词，这样不

被索引的词叫做停用词。Analyzer 可能会去掉一些停用词。

```
public class NgramAnalyzer extends Analyzer {
    @Override
    protected TokenStreamComponents createComponents(String fieldName,
                                                     Reader reader) {
        BigramDictioanry dict = BigramDictioanry.getInstance("./dic/");
        Tokenizer source = new NgramTokenizer(reader,dict);
        return new TokenStreamComponents(source);
    }
}
```

然后需要 Tokenizer 的工厂类 CnTokenizerFactory 来封装 Tokenizer 的创建细节。

```
public class CnTokenizerFactory extends TokenizerFactory implements
        ResourceLoaderAware {
    public CnTokenizerFactory(Map<String, String> args) {
        super(args);                          //给超类构造函数传入参数
        dicPath = args.get("dicDir");
    }
    static BigramDictioanry dict;             //分词词典
    private String dicPath;

      @Override
      public Tokenizer create(AttributeFactory factory, Reader input) {
          if (dicPath != null && dict==null) {
             dict = BigramDictioanry.getInstance(dicPath);
            }
          return new NgramTokenizer(factory,input, dict);
      }
    @Override
    public void inform(ResourceLoader loader) {
        if (dicPath != null) {
            System.out.println("词典路径=" + dicPath);
            dict = BigramDictioanry.getInstance(dicPath);
        } else {
            System.out.println ("未设置词典路径");
        }
    }
}
```

2.1.3 概率语言模型的分词方法

两个词可以组合成一个词的情况叫做组合歧义，如“上海/银行”“上海银行”，最大长度匹配算法无法正确切分组合歧义。例如，会把“请在一米线外等候”错误地切分成“一/米线”而不是“一/米/线”。

对于输入字符串 C“有意见分歧”，有下面两种切分可能：

S_1：有/ 意见/ 分歧/
S_2：有意/ 见/ 分歧/

这两种切分方法分别叫做 S_1 和 S_2，如何选择这两个切分方案？哪个切分方案更有可

能在语料库中出现就选择哪个切分方案。

可以计算条件概率 $P(S_1|C)$和 $P(S_2|C)$，然后根据 $P(S_1|C)$和 $P(S_2|C)$的值来决定选择 S_1 还是 S_2。

因为联合概率 $P(C,S) = P(S|C) \times P(C) = P(C|S) \times P(S)$，所以有：

$$P(S \mid C) = \frac{P(C \mid S) \times P(S)}{P(C)}$$

上面的公式也叫做贝叶斯公式，$P(C)$是字串在语料库中出现的概率。比如说语料库中有 10,000 个句子，其中有一句是 “有意见分歧” 那么 $P(C)$=P("有意见分歧")=1/10000。

在贝叶斯公式中，$P(C)$只是一个用来归一化的固定值，所以实际分词时并不需要计算。

$P(C|S)$是由从词串 S 恢复到汉字串 C 的概率，该值为 1，即 $P(C|S_1)$=$P(C|S_2)$=1。因此，比较 $P(S_1|C)$和 $P(S_2|C)$的大小，变成比较 $P(S_1)$和 $P(S_2)$的大小。也就是说：

$$\frac{P(S_1 \mid C)}{P(S_2 \mid C)} = \frac{P(S_1)}{P(S_2)}$$

因为 $P(S_1)$=P(有,意见,分歧)> $P(S_2)$=P(有意,见,分歧)，所以选择切分方案 S_1 而不是 S_2。

在具体的分词过程中，输入是一个字串 C=C_1,C_2,C_n，输出是一个词串 S=W_1,W_2,…,W_m，其中 $m \leqslant n$。对于一个特定的字符串 C，会有多个切分方案 S 对应，分词的任务就是在这些 S 中找出一个切分方案 S，使得 $P(S|C)$的值最大，$P(S|C)$就是由字符串 C 产生切分 S 的概率。最可能的切分方案为：

$$\begin{aligned}\text{BestSeg}(c) &= \arg\max_{S \in G} P(S \mid C) = \arg\max_{S \in G} \frac{P(C \mid S)P(S)}{P(C)} \\ &= \arg\max_{S \in G} P(S) = \arg \max_{w_1, w_2, \ldots, w_m \in G} P(w_1, w_2, \cdots, w_m)\end{aligned}$$

也就是对输入字符串切分出最有可能的词序列。

这里的 G 表示切分词图。待切分字符串 C 中的某个子串构成一个词 W，把这个词看成是从开始位置 i 到结束位置 j 的一条有向边。把 C 中的每个位置看成点，词看成边，可以得到一个有向图，这个图就是切分词图 G。

概率语言模型分词的任务是：在全切分所得的所有结果中求某个切分方案 S，使得 $P(S)$为最大。那么，如何来表示 $P(S)$呢？为了简化计算，假设每个词之间的概率是上下文无关的，则

$$P(S) = P(Pw_1, w_2, ..., w_m) \approx P(w_1) \times P(w_2) \times \cdots \times P(w_m)$$

其中，$P(w)$就是词 w 出现在语料库中的概率。例如：

$$P(S_1) = P(\text{有, 意见, 分歧}) \approx P(\text{有}) \times P\text{意见}) \times P(\text{分歧})$$

对于不同的 S，m 的值是不一样的，一般来说 m 越大，$P(S)$值越小。也就是说，分出的词越多，概率越小。这符合实际的观察，如最大长度匹配切分往往会使得 m 较小。

词表中的词往往很多，分摊到一个词的概率可能很小，所以 $P(S)$一般是通过很多小数值的连乘积算出来的，如果一个数太小，可能会向下溢出变成 0。例如 0.00000000000000

0000000000000001，double 类型表示不出如此小的数。因为函数 $y=\log(x)$，当 x 增大，y 也会增大，所以是单调递增函数，取 log 后，表示一个小于 1 的正数的精确度加大了。

$P(S)\approx P(w_1)\times P(w_2)\times\cdots\times P(w_m)\propto \log P(w_1)+\log P(w_2)+\cdots+\log P(w_m)$ 这里的$\propto$是正比符号。因为词的概率小于 1，所以取 log 后是负数，最后算 $\log P(w)$。

计算任意一个词出现的概率如下：

$$P(w_i)=\frac{w_i\text{在语料库中的出现次数}n}{\text{语料库中的总词数}N}$$

因此

$$\log P(w_i)=\log(Freq_w)-\log N$$

如果词概率的对数值事前已经算出来了，则结果直接用加法就可以得到 $\log P(S)$，而加法比乘法速度更快。

这个计算 $P(S)$的公式也叫做基于一元概率语言模型的计算公式，这种分词方法简称一元分词，它综合考虑了切分出的词数和词频。一般来说，词数少，词频高的切分方案概率更高。假设有一种特殊的情况：当所有词的出现概率相同时，则一元分词退化成最少词切分方法。

句子切分的准确性在很大程度上取决于词语的上下文。比如，“上海银行间的拆借利率上升”，“上海银行”后接词为“间”，这决定了“上海银行”应该切分为两个词“上海”和“银行”，而不是一个专有名词“上海银行”。

在一元分词中假设前后两个词的出现概率是相互独立的，在实际中不太可能。语言学家认为，一个词语的含义取决于它周围的词语，也就是说，某些词语会以很大概率经常出现在一起，比如，甜品店附近经常有咖啡馆，所以这两个词是正相关。但是很少会有人把“甜品店”和“沙县小吃”相提并论，不过“羡慕”“嫉妒”“恨”这三个词有时候会连续出现。切分出来的词序列越通顺，越有可能是正确的切分方案。N 元模型使用 n 个单词组成的序列来衡量切分方案的合理性。比如，估计单词 w_1 后出现 w_2 的概率，根据条件概率的定义：

$$P(w_2\mid w_1)=\frac{P(w_1,w_2)}{P(w_1)}$$

可以得到 $P(w_1,w_2)=P(w_1)\,P(w_2|w_1)$

同理 $P(w_1,w_2,w_3)=P(w_1,w_2)\,P(w_3|w_1,w_2)$

所以有 $P(w_1,w_2,w_3)=P(w_1)\,P(w_2|w_1)\,P(w_3|w_1,w_2)$

一般的形式为：

$$P(S)=P(w_1,w_2,\cdots,w_n)=P(w_1)P(w_2|w_1)P(w_3|w_1,w_2)\cdots P(w_n|w_1w_2\cdots w_{n-1})$$

这叫做概率的链规则。其中，$P(w_2|w_1)$表示 w_1 之后出现 w_2 的概率。如果词 w_1 和 w_2 独立出现，则 $P(w_2|w_1)$等价于 $P(w_2)$。

这样需要考虑在 n-1 个单词序列后出现的单词 w 的概率。直接使用这个公式计算 $P(S)$会存在两个致命的缺陷：一是参数空间过大，不可能实用化；另一个是数据稀疏严重。

例如，词汇量（V）=20,000 时，可能的二元（bigrams）组合数量有 400,000,000 个；可能的三元（trigrams）组合数量有 8,000,000,000,000 个；可能的四元（4-grams）组合数量有 1.6×10^{17} 个。

为了解决这个问题，我们引入了马尔可夫假设：一个词的出现仅仅依赖于它前面出现的有限的一个或者几个词。

如果简化成一个词的出现仅依赖于它前面出现的一个词，那么就称为二元语言模型（Bigram），即

$$P(S)=P(w_1,w_2,...,w_n)=P(w_1)\,P(w_2|w_1)\,P(w_3|w_1,w_2)\ldots P(w_n|w_1w_2\ldots w_{n-1})$$
$$\approx P(w_1)\,P(w_2|w_1)P(w_3|w_2)\cdots P(w_n|w_{n-1})$$

例如，$P(S_1)=P(\text{有})\,P(\text{意见}|\text{有})\,P(\text{分歧}|\text{意见})$

如果简化成一个词的出现仅依赖于它前面出现的两个词，那么就称之为三元语言模型（Trigram）。如果一个词的出现不依赖于它前面出现的词，则叫做一元语言模型（Unigram），也就是已经介绍过的概率语言模型的分词方法。

如果切分方案 S 是 n 个词组成的序列，那么 $P(w_1)\ P(w_2|w_1)P(w_3|w_2)\cdots P(w_n|w_{n-1})$也是 n 项连乘积。语言模型无论采用一元、二元还是三元都是 n 项连乘积，只不过二元以上模型是条件概率的连乘积。例如，对于切分“产品和服务”来说，二元模型计算：$P(\text{产品})\,P(\text{和}|\text{产品})\,P(\text{服务}|\text{和})$，三元模型计算：$P(\text{产品})\,P(\text{和}|\text{产品})\,P(\text{服务}|\text{产品},\text{和})$。

因为 $P(w_i|w_{i-1})=\text{freq}(w_{i-1},w_i)\ /\text{freq}(w_{i-1})$，所以二元分词不仅用到二元词典，还需要用到一元词典。

概率语言模型中文分词切分过程说明如下。

（1）把输入字符串切分成句子：对一段文本进行切分，依次从这段文本中切分出一个个句子，然后对句子逐个进行分词。

（2）原子切分：对于一个句子的切分，首先是通过原子切分，将整个句子切分成一个个的原子单元（即不可再切分的形式，例如 Java 这样的英文单词可以看成不可再切分的）。

（3）生成 n 元切分词图：根据基本词库对句子进行全切分，并且生成一个邻接链表表示的基本词图，再根据基本词图得到 n 元词图。

（4）计算最佳切分路径：在这个词图的基础上，运用动态规划算法找出切分最佳路径。

（5）按 Lucene 和 Elasticsearch 定义的 API 输出结果。

首先执行原子切分，处理中文串中的数字或者英文字符串，然后执行二元概率切分。主要代码如下：

```
//原子切分
fstSeg.seg(sentence,g);
//二元分词
GraphFactory.seg(sentence, g);
```

二元切分词图简称二元词图，n 元切分词图简称 n 元词图。我们可以考虑如何得到二元词图。一个词的开始位置和结束位置组成的节点组合是二元词图中的点，前后两个词的转移概率作为边的权重。

“有意见分歧”这句话中节点的组合有：{0,1}、{0,2}、{1,2}、{1,3}、{2,3}、{3,4}、{3,5}，得到的二元切分词图如图 2-4 所示。

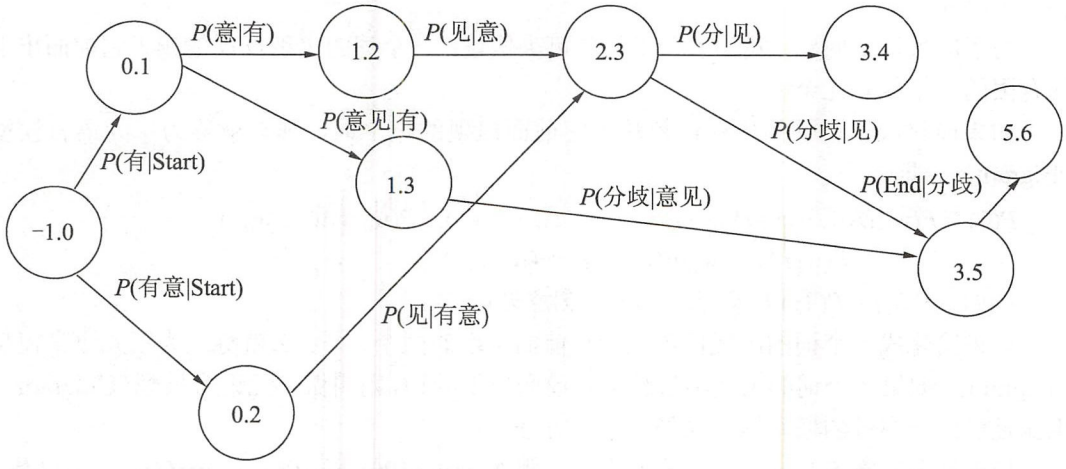


图 2-4　二元分词词图

生成词图，代码如下：

```
public class LatticeFactory {
    public static TernarySearchTrie dic = null;     //词典树
    static FSTSGraph fstSeg;                        //有限状态转换

    static {
        try {
            fstSeg = new FSTSGraph();
        } catch (Exception e) {
            e.printStackTrace();
        }

        dic =(new DicFileFactory()).create();       //创建词典树
    }

    public static AdjList getLattice(String text) throws Exception{
        int sLen = text.length();                   // 字符串长度
        AdjList g = new AdjList(sLen + 2);          //存储所有被切分的可能的词

        //原子切分
        SegScheme schema = fstSeg.seg(text);

        //用来存放前驱词的集合
        ArrayList<WordEntry> prevWords = new ArrayList<WordEntry>();

        //从前往后求出每个节点的最佳前驱节点和它的节点概率
        int fromPoint = 0;
        while (fromPoint >= 0) {
            int start = schema.startPoints.nextSetBit(fromPoint);
```

```
                                                    //开始点
            int end = schema.endPoints.nextSetBit(fromPoint + 1);
                                                    //结束点

            fromPoint = schema.startPoints.nextSetBit(start + 1);

            //从词典中查找前驱词的集合
            boolean match = dic.matchAll(text, start, prevWords,schema.
            endPoints);

            if (!match) {
                //词典中找不到对应的词，则返回开始点和结束点之间的字符串
                String word = text.substring(start, end);
                prevWords.add(new WordEntry(word, 1));
                Node newEdge =
                    new Node(start,start+word.length(),Math.log
                    ((double)1/dic.n));
                g.addEdge(newEdge);                 //词图增加边
            } else{
                for(WordEntry w:prevWords){         //遍历找到的每个词
                    Node newEdge =
                      new Node(start,start+w.word.length(),Math.log
                      ((double)w.freq/dic.n));
                    g.addEdge(newEdge);             //词图增加边
                }
            }
        }

        return g;
    }
}
```

二元分词方法切分文本的代码如下：

```
public static List<String> split(String text) throws Exception {
    AdjList wordGraph = LatticeFactory.getLattice(text);  //得到词图
    bestPrev(wordGraph);                                  //从后往前计算最佳前驱节点
    ArrayDeque<Node> seq = new ArrayDeque<Node>(); // 切分出来的节点序列
    //从后向前找最佳前驱节点
    for (Node t = wordGraph.endNode.bestPrev; t.start > -1; t = t.bestPrev){
        seq.addFirst(t);
    }
    return bestPath(text, seq);
}
```

从后往前计算词图中每个节点的最佳前驱节点：

```
public static AdjList bestPrev(AdjList wordGraph) throws Exception {
    for (Node currentNode : wordGraph) { //从前往后遍历切分词图中的每个节点

        // 得到当前节点的前驱节点集合
        NodeLinkedList prevNodes = wordGraph.prevNodes(currentNode);
```

```
            double nodeProb = minValue;                    //侯选词概率
            Node minNode = null;
            if (prevNodes == null) {
                currentNode.nodeProb = 0;
                continue;
            }
            for (Node prevNode : prevNodes) {
                double currentProb = transProb(prevNode, currentNode)
                                                           //转移概率
                        + prevNode.nodeProb;               //前一个节点的节点概率

                if (currentProb > nodeProb) {
                    nodeProb = currentProb;
                    minNode = prevNode;
                }
            }
            currentNode.bestPrev = minNode;                // 设置当前词的最佳前驱词
            currentNode.nodeProb = nodeProb;               // 设置当前词的词概率
        }

        return wordGraph;
    }
```

然后计算节点之间的转移概率，代码如下：

```
static double lambda1 = 0.3;                               //平滑参数
static double lambda2 = 0.7;
// 前后两个词的转移概率
private static double transProb(Node prevWord, Node currentWord) {
    double transProb = lambda1 * currentWord.frq / LatticeFactory.dic.n +
lambda2
            * (bigramFreq / prevWord.frq);                 //平滑后的二元概率

    return transProb;                                      //得到转移概率
}
```

测试分词，代码如下：

```
String sentence = "巨星和苯磺隆为高效内吸药剂，一般施药后一周左右杂草开始枯黄死亡。";
Segmenter seg = new Segmenter();
List<WordTokenInf> words = seg.split(sentence);
for (WordTokenInf word : words) {
    System.out.print(word.termText + " ");
}
```

为了更准确地搜索，可以标注词性，为了方便指明词的词性，可以给每个词性编码。例如，根据英文缩写，把“形容词”编码成 a，名词编码成 n，动词编码成 v...。如表 2-1 所示为完整的词性编码表。

给句子标注词性，例如“不/d 忘/v 群众/n 疾苦/n 温暖/v 送/v 进/v 万/m 家/q”。可以使用隐马尔可夫模型（Hidden Markov Model，HMM）实现词性标注。

表 2-1　词性编码表

代　码	名　称	举　例
11	形容词	最/d、大/a、的/u
ad	副形词	一定/d、能够/v、顺利/ad、实现/v、。/w
ag	形语素	喜/v、煞/ag、人/n
an	名形词	人民/n、的/u、根本/a、利益/n、和/c、国家/n、的/u、安稳/an、。/w
b	区别词	副/b、书记/n、王/nr、思齐/nr
c	连词	全军/n、和/c、武警/n、先进/a、典型/n、代表/n
d	副词	两侧/f、台柱/n、上/f、分别/d、雄踞/v、着/u
dg	副语素	用/v、不/d、甚/dg、流利/a、的/u、中文/nz、主持/v、节目/n、。/w
e	叹词	嗬/e、！/w
f	方位词	从/p、一/m、大/a、堆/q、档案/n、中/f、发现/v、了/u
g	语素	dg、ag
h	前接成分	目前/t、各种/r、非/h、合作制/n、的/u、农产品/n
i	成语	提高/v、农民/n、讨价还价/i、的/u、能力/n、。/w
j	简称略语	民主/ad、选举/v、村委会/j、的/u、工作/vn
k	后接成分	权责/n、明确/a、的/u、逐级/d、授权/v、制/k
l	习用语	是/v、建立/v、社会主义/n、市场经济/n、体制/n、的/u、重要/a、组成部分/l、。/w
m	数词	科学技术/n、是/v、第一/m、生产力/n
n	名词	希望/v、双方/n、在/p、市政/n、规划/vn
ng	名语素	就此/d、分析/v、时/Ng、认为/v
nr	人名	建设部/nt、部长/n、侯/nr、捷/nr
ns	地名	北京/ns、经济/n、运行/vn、态势/n、喜人/a
nt	机构团体	[冶金/n、工业部/n、洛阳/ns、耐火材料/l、研究院/n]nt
nx	字母专名	ＡＴＭ/nx、交换机/n
nz	其他专名	德士古/nz、公司/n
o	拟声词	汩汩/o、地/u、流/v、出来/v
p	介词	往/p、基层/n、跑/v、。/w
q	量词	不止/v、一/m、次/q、地/u、听到/v、，/w
r	代词	有些/r、部门/n
s	处所词	移居/v、海外/s、。/w
t	时间词	当前/t、经济/n、社会/n、情况/n
tg	时语素	秋/Tg、冬/tg、连/d、旱/a
u	助词	工作/vn、的/u、政策/n
ud	结构助词	有/v、心/n、栽/v、得/ud、梧桐树/n

（续）

代　码	名　称	举　例
ug	时态助词	你/r、想/v、过/ug、没有/v
uj	结构助词的	迈向/v、充满/v、希望/n、的/uj、新/a、世纪/n
ul	时态助词了	完成/v、了/ ul
uv	结构助词地	满怀信心/l、地/uv、开创/v、新/a、的/u、业绩/n
uz	时态助词着	眼看/v、着/uz
v	动词	举行/v、老/a、干部/n、迎春/vn、团拜会/n
vd	副动词	强调/vd、指出/v
vg	动语素	做好/v、尊/vg、干/j、爱/v、兵/n、工作/vn
vn	名动词	股份制/n、这种/r、企业/n、组织/vn、形式/n、，/w
w	标点符号	生产/v、的/u、５Ｇ/nx、/w、８Ｇ/nx、型/k、燃气/n、热水器/n
x	非语素字	生产/v、的/u、５Ｇ/nx、/w、８Ｇ/nx、型/k、燃气/n、热水器/n
y	语气词	已经/d、３０/m、多/m、年/q、了/y、。/w
z	状态词	势头/n、依然/z、强劲/a、；/w

2.1.4　中文分词插件原理

Lucene 处理同义词时需要把同义词表示为一个词图的形式。例如，为了支持按同义词搜索，当应用同义词：

```
domain name system → dns
```

到所处理的文本中：

```
domain name system is fragile
```

使用属性 PositionIncrementAttribute 和 PositionLengthAttribute 编码的词图如图 2-5 所示。

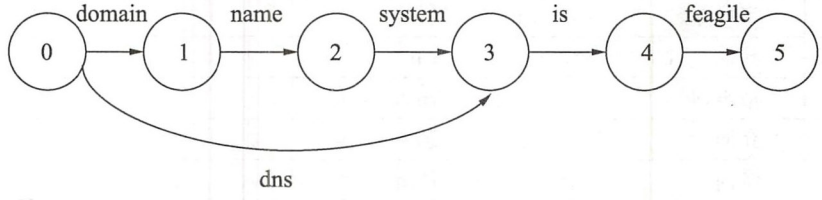


图 2-5　增加同义词 dns 的词图

然而，一旦将此文档添加到 Lucene 索引中，一些图结构就丢失了。因为 Lucene 忽略了 PositionLengthAttribute 属性，该属性会规定一个给定的符号何时终止，这样导致词图平坦化成图 2-6 所示。

这样，查询"dns is fragile"便无法匹配它应该匹配的文档。同样，查询"dns name system"不应该匹配到这个文档，但实际却会匹配上。

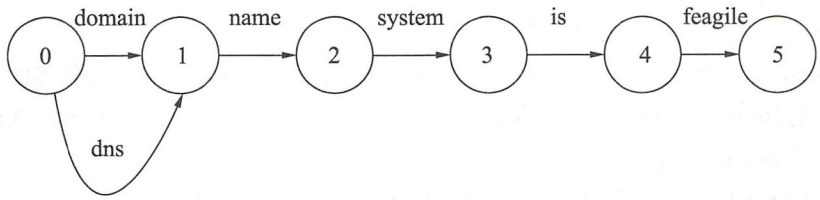


图 2-6　平坦化的词图

更糟糕的是，如果同义词插入了多个符号，例如，将上述折叠规则反转为一个扩展规则：

```
dns → domain name system
```

然后分析如下文本：

```
dns is fragile
```

那么即使是由 SynonymFilter 生成的符号，但在索引之前就已经被平坦化了！这是 Lucene 面临的同义词处理方面的问题。

Elasticsearch 5.2.0 中包含的 Lucene 6.4.0 有了让人满意的解决办法，可以进行同义词查询，只要在搜索时间而不是索引时间应用同义词即可。

Elasticsearch 5.2.0 第一个大的改变是增加了 SynonymGraphFilter 同义词过滤器替换过时的 SynonymFilter 同义词过滤器。这个新的同义词过滤器可以在任何情况下生成如图 2-4 所示的正确的图表示，不管你输入的是单个符号还是多个符号。此外，Elasticsearch 5.2.0 还有一个新的 FlattenGraphFilter 过滤器，可以压缩图符号流以便用于索引。如果确实需要旧的 SynonymFilter 一样的行为，那么可以在 SynonymGraphFilter 同义词过滤器后接一个过滤器 FlattenGraphFilter，但是 FlattenGraphFilter 过滤器有时会丢掉一些图结构。

Elasticsearch 5.2.0 改进的第二个方面是提高了查询解析器的效率。首先，旧的经典查询解析器当在空格时会终止解析预切分的输入查询文本。一定要记住调用 setSplitOnWhitespace(false)，因为空格切分是默认的设置以便向后兼容。这样就让查询时的解析器可以看见多个符号而不是把每个符号分开处理。这些对 QueryBuilder 中复杂逻辑的简化是一个重要的引导。

Elasticsearch 5.2.0 改进的第三方面是检查查询分析器何时产生图表示，并且生成合适的查询。现在查询分析器（也就是基类 QueryBuilder）可以监控 PositionLengthAttribute，当符号的 PositionLengthAttribute 值大于 1 时计算如图 2-4 所示的图中所有的路径。在 Elasticsearch 中，match_query、query_string 和 simple_query_string 查询都可以正确处理词图。

上述的改进方面可以让我们在查询时使用多符号同义词，并取得精确的结果。比如，如果查询是：

```
dns is fragile
```

如果没有引号，那么过滤器 SynonymGraphFilter 可以将 dns 扩展到 domain name system，然后查询分析器将分析查询建立整个图表示，并且计算整个图中的不同路径：

```
dns is fragile
domain name system is fragile
```

Elasticsearch 会分别分析上述两个字符串，产生两个子查询，并且使用 SHOULD 子句将它们组合在 BooleanQuery 中。如果原始查询拥有两个引号，则使用 PhraseQuery 将两者组合，产生出确定的答案。

在 Lucene 6.5.0 中，对查询器 QueryBuilder 的优化需要分析查询的图表示（关节点），以生成更有效的二值查询器，避免可能的路径组合维度过大。这种优化主要难点包括：如何存储图查询；如何将二值处理器应用到同义词扩展；属性 minShouldMatch 如何工作等。例如，如果用户没有加括号，但是插入多符号查询，该使用短语查询还是默认的查询解析器操作符呢？希望在图查询的实践经验中能找到答案。

除了同义词外，Lucene 还有更多的生成图表示的图过滤器。在 Lucene 6.5.0 版本中，WordDelimiterFilter 将会被替换成 WordDelimiterGraphFilter。针对日语的处理器 JapaneseTokenizer 基于 Kuromoji 日语形态分析器，已经可以生成词图来表示整个单词和可能的子单词。此外，还有其他的词过滤器，如 shingles、ngrams 等，它们都应该生成图输出，但是还没有修订。

为了处理多符号同义词，必须在查询时使用同义词处理器，而不是在索引时使用，因为 Lucene 的索引还不能存储词图表示。和索引时同义词处理相比，查询时的同义词处理需要更多的 CPU 和 I/O 工作量，因为需要访问更多的术语来回答问题，但是索引库小多了。查询时的同义词处理更灵活，因为当改变同义词时，不需要再次建立索引。

另外一个挑战是 SynonymGraphFilter，它在生成正确的图表示时，不能处理图表示，这意味着在使用 WordDelimiterGraphFilter 或 JapaneseTokenizer 后不能使用 SynonymGraphFilter，而且希望同义词会匹配输入图的片段。

2.1.5 开发中文分词插件

本节来写一个支持 Lucene 6.6.0 的分词器，项目中除了基本的 lucene-core-6.6.0.jar、lucene-codecs-6.6.0.jar、junit-4.10.jar 文件外，还需要添加 randomizedtesting-runner-2.5.2.jar 和 lucene-test-framework-6.6.0.jar 这两个.jar 文件。

如果使用 Maven，则需要添加依赖关系：

```
<dependency>
    <groupId>org.apache.lucene</groupId>
        <artifactId>lucene-test-framework</artifactId>
    <version>6.6.0</version>
    <scope>test</scope>
</dependency>
```

测试中文分词 Tokenizer：

```
String sentence = "我们胜利了";
StringReader input = new StringReader(sentence);
```

```
BigramDictioanry dict = BigramDictioanry.getInstance("./dic/");
BigramTokenizer tokenizer = new BigramTokenizer(dict);
tokenizer.setReader(input);
tokenizer.reset();
while (tokenizer.incrementToken()) {
    CharTermAttribute termAtt = tokenizer
            .getAttribute(CharTermAttribute.class);
    System.out.println("term:" + termAtt);      // 打印词
}
tokenizer.close();
```

测试类需要生成随机值，用所有可能的数据测试用户的代码，以便将来用户的代码能够处理任何类型的数据。RandomizedTesting 是一个随机测试基础设施，使用 RandomizedTest.randomInt()方法可得到随机整型值。下面是一个使用 RandomizedTest 的例子。

```
public class TestUsingRandomness extends RandomizedTest {

  @Test
  public void expectNoException() {
    String [] words = {"oh", "my", "this", "is", "bad."};

    //从上面的数组中选出一个随机词
    System.out.println(words[Math.abs(randomInt()) % words.length]);
  }
}
```

MockAnalyzer 和 MockTokenizer 用于分析相关的测试词语，相关的测试代码如下：

```
public class TestIndex extends RandomizedTest {

    public static void test1() throws IOException {
        Analyzer analyzer = new MockAnalyzer(LuceneTestCase.random(),
                MockTokenizer.SIMPLE, true);
        Directory rd = new RAMDirectory();
        IndexWriter w = new IndexWriter(rd, new IndexWriterConfig
        (analyzer));
    }

}
```

增加引用的 jar 包：lucene-analyzers-common-6.6.0.jar。实现中文分词的 TokenizerFactory 工厂类 BigramTokenizerFactory 代码如下：

```
public class BigramTokenizerFactory extends TokenizerFactory{

    protected BigramTokenizerFactory(Map<String, String> args) {
        super(args);
    }

    @Override
    public Tokenizer create(AttributeFactory factory) {
        String dicPath = "./dic/";
        BigramDictioanry dict = BigramDictioanry.getInstance(dicPath);
        return new BigramTokenizer(factory,dict);
    }

}
```

BaseTokenStreamTestCase 是所有使用 TokenStreams 的 Lucene 单元测试的基类。使用 BaseTokenStreamTestCase 测试 BigramTokenizerFactory 的代码如下：

```
public class TestBigramTokenizerFactory extends BaseTokenStreamTestCase{
     public void testSimple() throws Exception {
       Reader reader = new StringReader("我购买了道具和服装。");
       TokenizerFactory factory =
          new BigramTokenizerFactory(new HashMap<String,String>());
       Tokenizer tokenizer = factory.create(newAttributeFactory());
       tokenizer.setReader(reader);
       assertTokenStreamContents(tokenizer,
          new String[] { "我", "购买", "了", "道具", "和", "服装", "。" });
    }
}
```

编写支持词性标注的 NgramAnalyzer：

```
package com.lietu.bigramSeg;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.Tokenizer;
import org.apache.lucene.util.AttributeFactory;

public class NgramAnalyzer extends Analyzer {
    BigramDictioanry dict;
    Tagger tagger;

    public NgramAnalyzer(String p) {
        init(p);
    }

    public NgramAnalyzer() {
        String path = "./dic/";
        init(path);
    }

    public void init(String dicPath){
        dict = BigramDictioanry.getInstance(dicPath);
        tagger = Tagger.getInstance(dicPath);
    }

    @Override
    protected TokenStreamComponents createComponents(String arg0) {
        Tokenizer source = new NgramTagnizer(
                AttributeFactory.DEFAULT_ATTRIBUTE_FACTORY,
dict,tagger);
        return new TokenStreamComponents(source);
    }
}
```

可以增加语义规则提高切分准确度。例如，根据地名的词性编码 ns，得到语义规则"去<ns>"。整合语义规则的分词实现代码如下：

```
RuleSegmenter seg = new RuleSegmenter();
String word = "北京";
String mean = "ns";
```

```
seg.dic.addWord(word, mean);

String pattern = "去<ns>";  // 去+地名
seg.addRule(pattern);

String text = "提出去北京";

AdjList g = seg.combineSuc(text);

ArrayDeque<Integer> path = seg.bestPath(g);

// 输出切分结果
int start = 0;
for (Integer end : path) {
    System.out.print(text.substring(start, end) + "/ ");
    start = end;
}
```

2.1.6　支持 Elasticsearch 的插件

本节参考 IK 分词插件（网址为 https://github.com/medcl/elasticsearch-analysis-ik）使用 Maven（m2eclipse）建立一个项目。

在 Eclipse 中显示这个错误如下：

```
Description Resource Path Location Type Could not calculate build plan:
Failure to transfer org.apache.maven.plugins:maven-compiler-plugin:
pom:2.0.2from http://repo1.maven.org/maven2 was cached in the local
repository, resolution will not be reattempted until the update interval
of central has elapsed or updates are forced. Original error: Could not
transfer artifact org.apache.maven.plugins:maven-compiler-plugin:pom:
2.0.2 from/to central (http://repo1.maven.org/maven2): No response
received after 60000 ExampleProject Unknown Maven Problem.
```

找到{user}/.m2/repository 目录，在窗口右上角的“搜索”栏中输入“.lastupdated”，在 repository 目录中查看这些文件的所有子文件夹。通过右击并选择“删除”命令来删除它们。然后回到 Eclipse 中，右击项目并选择 Maven|“更新项目”命令，在弹出的对话框中选择“Force Update of Snapshots/Releases”，单击“确定”按钮，依赖关系终于可以正确解析。

首先创建仅包含一个 PLUGIN_NAME 属性的类 AnalysisBgsegPlugin，然后编写支持中文的 AnalyzerProvider 类。代码如下：

```
package org.elasticsearch.index.analysis;

import java.nio.file.Path;

import org.elasticsearch.common.settings.Settings;
import org.elasticsearch.env.Environment;
import org.elasticsearch.index.IndexSettings;
import org.elasticsearch.plugin.analysis.bgseg.AnalysisBgsegPlugin;

import com.lietu.bigramSeg.NgramAnalyzer;
```

```
public class BigramAnalyzerProvider extends AbstractIndexAnalyzerProvider
<NgramAnalyzer> {
    private final NgramAnalyzer analyzer;

    public BigramAnalyzerProvider(IndexSettings indexSettings,
    Environment env, String name, Settings settings) {
        super(indexSettings, name, settings);
        //得到词典路径
        Path dicPath = env.configFile().resolve(AnalysisBgsegPlugin.
        PLUGIN_NAME);

        analyzer=new NgramAnalyzer(dicPath.toString());
    }

    @Override public NgramAnalyzer get() {
        return this.analyzer;
    }
}
```

在 Elasticsearch 中使用的 TokenizerFactor 工厂类 BigramESTokenizerFactory：

```
package org.elasticsearch.index.analysis;

import org.apache.lucene.analysis.Tokenizer;
import org.elasticsearch.common.settings.Settings;
import org.elasticsearch.env.Environment;
import org.elasticsearch.index.IndexSettings;
import org.elasticsearch.plugin.analysis.bgseg.AnalysisBgsegPlugin;

import com.lietu.bigramSeg.BigramDictioanry;
import com.lietu.bigramSeg.BigramTokenizer;

import java.nio.file.Path;

public class BigramESTokenizerFactory extends AbstractTokenizerFactory {
    private Path dicPath;                          //词典路径

    public BigramESTokenizerFactory(IndexSettings indexSettings,
    Environment env,
            String name, Settings settings) {
        super(indexSettings, name, settings);
        dicPath = env.configFile().resolve(AnalysisBgsegPlugin.
        PLUGIN_NAME);
    }

    @Override
    public Tokenizer create() {
        //得到二元词典
        BigramDictioanry dict = BigramDictioanry
                .getInstance(dicPath.toString());
        return new BigramTokenizer(dict);
    }

}
```

完善插件类 AnalysisBgsegPlugin：

```
package org.elasticsearch.plugin.analysis.bgseg;

import java.util.HashMap;
import java.util.Map;

import org.apache.lucene.analysis.Analyzer;
import org.elasticsearch.index.analysis.AnalyzerProvider;
import org.elasticsearch.index.analysis.BigramAnalyzerProvider;
import org.elasticsearch.index.analysis.BigramESTokenizerFactory;
import org.elasticsearch.index.analysis.TokenizerFactory;
import org.elasticsearch.indices.analysis.AnalysisModule;
import org.elasticsearch.plugins.AnalysisPlugin;
import org.elasticsearch.plugins.Plugin;

public class AnalysisBgsegPlugin extends Plugin implements AnalysisPlugin {

    public static String PLUGIN_NAME = "analysis-bgseg";

    @Override
    public Map<String, AnalysisModule.AnalysisProvider
    <TokenizerFactory>> getTokenizers() {
        Map<String, AnalysisModule.AnalysisProvider<TokenizerFactory>>
        extra = new HashMap<>();

        //通过构造方法得到 Tokenizer 工厂类
        extra.put("bs_max_word", BigramESTokenizerFactory::new);

        return extra;
    }

    @Override
    public Map<String, AnalysisModule.AnalysisProvider<AnalyzerProvider
    <? extends Analyzer>>> getAnalyzers() {
        Map<String, AnalysisModule.AnalysisProvider<AnalyzerProvider<?
        extends Analyzer>>> extra = new HashMap<>();
        //通过构造方法得到 AnalyzerProvider 类
        extra.put("bs_max_word", BigramAnalyzerProvider::new);

        return extra;
    }
}
```

2.1.7　中文分析器提供者

可以把生成的 elasticsearch-analysis-bs-5.5.0.jar 文件放入插件路径下 D:\elasticsearch-5.5.0 \plugins\ bs，或者使用 elasticsearch-plugin 来安装：

```
./bin/elasticsearch-plugin  install  elasticsearch-analysis-bs-5.5.0.zip
```

在 elasticsearch.yml 配置文件中增加中文分析器：

```
index:
  analysis:
    analyzer:
      cn:
          alias: [cn_analyzer]
          type: org.elasticsearch.index.analysis.BigramAnalyzerProvider

index.analysis.analyzer.default.type
     : " org.elasticsearch.index.analysis.BigramAnalyzerProvider "
```

可以使用 CURL 测试分析器：

```
# curl -XGET 'localhost:9200/_analyze?pretty' -H 'Content-Type:
application/json' -d'
{
  "analyzer" : "standard",
  "text" : "this is a test"
}'
```

也可以在 head 插件中测试这个分析器，head 插件中的索引选项下面有个 test analyze 选项。设置全局默认分词会看到按词切分的效果，如果没有设置，则还是会按照 Elasticsearch 默认的一元分词来处理。可以先创建一个索引库，然后在这个索引上测试分词器。

查看分词效果的命令如下：

```
_analyze?text='我爱北京天安门'&analyzer=standard
_analyze?text='我爱北京天安门'&analyzer=cn
```

例如，news 索引使用如下的测试地址：

```
http://localhost:9200/news/_analyze?analyzer=cn&text='我爱北京天安门'
http://localhost:9200/news/_analyze?analyzer=standard&text='我爱北京天安门'
```

```
inquisitor(https://github.com/polyfractal/elasticsearch-inquisitor)
是一个测试分词的插件
```

index_analyzer 代表这个字段建立索引时使用的分词方式，search_analyzer 代表对这个字段搜索时使用的分词。

可以使用 Mappings 在索引中指定切分方式。示例如下：

```
{
   "recruitinfo":{
      "properties":{
         "id":{
            "type":"string",
            "index":"not_analyzed"
         },
         "title":{
            "type":"string",
            "term_vector": "with_positions_offsets",
            "index_analyzer": "cn",
            "search_analyzer": "cn",
            "store":"yes"
         }
```

```
        }
    }
}
```

使用这个 Mappings：

```
client.admin().indices().prepareCreate(indexName)
.setSettings("... your JSON settings..")
   .addMapping(type, "... your mapping...")
```

2.1.8　字词混合索引

查询某些短语时，按字分词列和按词分词列的返回结果数量是不一样的。测试代码如下：

```
public static void searchField(String field,String keyWords ){
    MatchQueryBuilder qb = QueryBuilders.matchPhraseQuery(field,
    keyWords);

    Client client = getClient();
    SearchResponse searchResponse = client
            .prepareSearch(ESConfig.indexName).setQuery(qb).execute()
            .actionGet();
    SearchHits hits = searchResponse.getHits();

    long totalHits = hits.getTotalHits();              //得到结果总数
    System.out.println("totalHits:" + totalHits);
}
```

分别使用按字分词列和按词分词列执行搜索：

```
String keyWords = "自定义分词器";
String field = "contentsS";
searchField(field ,keyWords);                          //按字查询

field = "contents";
searchField(field ,keyWords);                          //按词查询
```

输入相同的查询词后返回的结果数量不一样。

为了保证搜索的查全和查准性，对全文查询列采用单字索引和词索引两列都索引同样内容的方法。对于标题，按字索引的列在 Mapping 中定义如下：

```
        "stitle":{
           "type":"string",
           "term_vector": "with_positions_offsets",
           "index_analyzer": "standard",
           "search_analyzer": "standard",
           "store":"yes"
        },
```

按词索引的列在 Mapping 中定义如下：

```
        "title":{
           "type":"string",
           "term_vector": "with_positions_offsets",
           "index_analyzer": "cn",
```

```
            "search_analyzer": "cn",
            "store":"yes"
        }
```

这里的 cn 是在配置文件中指定的，standard 是 Elasticsearch 自带的。

打开 Index Metadata，可以看到索引库的结构。我们可以用 Java 代码修改索引库的结构，增加字索引列。

定义 Mapping，一列是按字索引，另外一列是按词索引。D:\elasticsearch-5.5.0\config\mappings\news 目录下的 type1.json 内容如下：

```
{
   "type1":{
      "properties":{
         "title":{
            "type":"string",
            "term_vector": "with_positions_offsets",
            "index_analyzer": "cn",
            "search_analyzer": "cn",
            "store":"yes"
         },
         "body":{
            "type":"string",
            "term_vector": "with_positions_offsets",
            "index_analyzer": "cn",
            "search_analyzer": "cn",
            "store":"yes"
         },
         "stitle":{
            "type":"string",
            "term_vector": "with_positions_offsets",
            "index_analyzer": "standard",
            "search_analyzer": "standard",
            "store":"yes"
         },
         "sbody":{
            "type":"string",
            "term_vector": "with_positions_offsets",
            "index_analyzer": "cn",
            "search_analyzer": "cn",
            "store":"yes"
         }
      }
   }
}
```

使用 API 创建 JSON 内容：

```
XContentBuilder mapping = XContentFactory
        .jsonBuilder()
        .startObject()
        .startObject(indexType)
        .startObject("properties")
        .startObject("title")
        .field("type", "string")
```

```
        // start title
        .field("store", "yes")
        .field("analyzer", "cn")                          //词
        .endObject()
        // end title
        .startObject("postDate")
        .field("type", "date").field("store", "yes")
        .field("index", "analyzed")
        .endObject()
        // end post date
        .startObject("body")
        .field("type", "string").field("store", "yes")
        .field("index_analyzer", "standard")              //字
        .field("search_analyzer", "standard")
        .endObject() // end field
        .endObject() // end properties
        .endObject() // end index type
        .endObject();
```

为了保证连续匹配的文档得分高，我们把短语查询和普通的模糊查询组合成布尔查询。写法如下：

```
//短语查询
MatchQueryBuilder pqTitle = QueryBuilders.matchPhraseQuery(title,
qString);
MatchQueryBuilder pqTitle2 = QueryBuilders.matchPhraseQuery(title2,
qString);
MatchQueryBuilder pqBody = QueryBuilders.matchPhraseQuery(body, qString);
MatchQueryBuilder pqBody2 = QueryBuilders.matchPhraseQuery(body2,
qString);
//普通模糊查询
QueryStringQueryBuilder fuzzyQb = new QueryStringQueryBuilder(qString)
      .field(title2).field(body2).field(title).field(body);
//把上面的查询组合成布尔查询得到最终的查询
QueryBuilder qb =
  queryBuilders.boolQuery().should(pqTitle2).should(pqTitle).
          should(pqBody2).should(pqBody).should(fuzzyQb);
```

2.2　提高分词准确度

因为 *n* 元连接的稀疏性，所以可以采用爬虫补充 *n* 元连接。从必应词典获取二元连接的代码如下：

```
public static Has

hMap<Bigram, int[]> getPairsByWordAndOffset(String word,
                  int pageOffet,
                   CloseableHttpClient httpclient) throws Exception {
    String urlAjax = "http://cn.bing.com/dict/service?q="
           + URLEncoder.encode(word, "UTF-8") + "&offset="
           + URLEncoder.encode(pageOffet + "", "UTF-8") + "&dtype=sen";
    HttpGet httpgetAjax = new HttpGet(urlAjax);
```

```
        HttpResponse responseAjax = httpclient.execute(httpgetAjax);

        HttpEntity entityAjax = responseAjax.getEntity();

        HashMap<Bigram, int[]> table = new HashMap<Bigram, int[]>();

        if (entityAjax != null) {
            // 读入内容流并以字符串形式返回，这里指定网页编码是 UTF-8
            // 网页的 Meta 标签中指定了编码
            String content = EntityUtils.toString(entityAjax, "utf-8");
            // System.out.println(content);
            Document doc = Jsoup.parse(content);
            Elements elementsLi = doc.select(".se_li");
            if (elementsLi == null || elementsLi.size() == 0) {
                return null;
            }
            // Jsoup 处理提取目标内容
            for (Element pairs : elementsLi) {

                if (pairs.select(".se_li1").size() == 0) {
                    continue;
                }
                Element s = pairs.select(".se_li1").first();

                if (s.select(".sen_en").size() == 0
                        || s.select(".sen_cn").size() == 0) {
                    continue;
                }
                Element senCn = s.select(".sen_cn").first();
                Elements wordEs = senCn.children();
                String preWord = null;

                for (Element w : wordEs) {
                    if (preWord == null) {
                        preWord = w.text();
                        continue;
                    }
                    String curWord = w.text();
                    Bigram bigram = new Bigram(preWord, curWord);

                    Object biItem = table.get(bigram);
                    if (biItem != null) {
                        ((int[]) biItem)[0]++;
                    } else {
                        int[] count = { 1 };
                        table.put(bigram, count);
                    }
                    preWord = curWord;
                }
            }
            EntityUtils.consume(entityAjax);                    //关闭内容流
        }

        return table;
    }
```

参考其他数据来源，筛选出有效的二元连接并存入数据库。

```
String insertSQL = "insert into bigram(prev,next)values(?,?)";
QueryRunner runner = new QueryRunner();


for (Entry<Bigram, int[]> e : table.entrySet()) {
    Bigram key = e.getKey();

    if (bigrams.contains(key)) {
        runner.update(conn, insertSQL, key.prev,
                key.next);
    }
}
```

2.3　本章小结

本章重点介绍了 *n* 元概率语言模型的分词插件开发与使用。为了使用 Elasticsearch 准确地搜索中文，除了使用字词混合索引的方法，还可以使用 IKAnalyzer。

对于亚洲其他国家的语言，如日文分词可以使用 Kuromoji（下载地址是 https://github.com/atilika/kuromoji）；韩文分词可以使用 Korean Analysis for ElasticSearch（下载地址是 https://github.com/usemodj/elasticsearch-analysis-korean）。

第 3 章　Mapping 详解

在使用数据库时，通过 DDL（Data Definition Language）定义模式（Schema），也就是一套相关的表结构。例如，定义一个模式中的新闻表如下：

```
CREATE  TABLE "news" (
  "url" string PRIMARY KEY  NOT NULL ,        --网址
  "title" text,                               --标题
  "body" text,                                --内容
  "pubdate" DATETIME)                         --发布时间
```

不同的字段类型有不同的操作方式。为了能够把日期字段处理成日期，把数字字段处理成数字，把字符串字段处理成全文本（Full-text）或精确的字符串值，Elasticsearch 需要知道每个字段是什么类型。

3.1　索引模式

Elasticsearch 中的索引模式叫做 Mapping。索引中的每个文档都有一个 type（类型），每个 type 拥有自己的模式或者模式定义（Schema Definition）。一个 Mapping 中定义的有字段类型、每个字段的数据类型，以及字段被 Elasticsearch 处理的方式。Mapping 还可用于设置关联到 type 上的元数据。

首先创建索引，然后再创建索引上的类型。例如，创建一个叫做 test 的索引，代码如下：

```
1  #curl -XPUT  http://localhost:9200/test -d'
2  {
3      "settings" : {
4          "index" : {
5              "number_of_shards" : 3,              //设定了索引的分片数量
6              "number_of_replicas" : 2             //副本数量
7          }
8      }
9  }'
```

3.1.1　创建模式

下面定义了一个搜索书籍的模式。首先，在 Linux 命令行下使用 Micro 编辑器编辑

mapping.json 文件：

```
# micro mapping.json
```

mapping.json 文件的内容如下：

```
{
    "book" : {
        "properties" : {
            "author" : {                          //定义字符串类型的作者列
                "type" : "string"
            },
            "title" : {                           //定义字符串类型的标题列
                "type" : "string"
            },
            "year" : {                            //定义长整型类型的年份列
                "type" : "long"
            },
            "available" : {                       //定义布尔类型的列，表示能否买到这本书
                "type" : "boolean",
                "index" : "analyzed"
            }
        }
    }
}
```

然后使用这个模式定义书籍索引结构：

```
#curl -XPUT 'http://localhost:9200/<indexname>/book/_mapping' -d
@mapping.json
```

返回结果如下，则表示成功执行。

```
{"acknowledged":true}
```

接着通过元字段"_parent"定义一个父子关系模式。使用 Micro 编辑器编辑 mapping_parent.json 文件：

```
# micro mapping_parent.json
```

mapping_parent.json 文件的内容如下：

```
{
    "book" : {                                    //定义图书索引库结构
        "properties" : {
            "title" : {                           //定义字符串类型的标题列
                "type" : "string"
            },
            "year" : {                            //定义长整型类型的年份列
                "type" : "long"
            },
            "available" : {                       //定义布尔类型的列，表示能否买到这本书
                "type" : "boolean",
                "index" : "analyzed"
            }
        }
    },
```

```
  "authors": {                                    //定义作者索引库结构
    "properties": {
      "first_name": { "type": "keyword" },        //定义关键词类型的 first_name 列
      "last_name": { "type": "keyword" }          //定义关键词类型的 last_name 列
    },
    "_parent": {                                  //声明作者索引库的父类型 books
      "type": "books"
    }
  }
 }
}
```

除了已经介绍过的当增加文档时需要填入数据的字段外，还有元字段（Meta-fields）。元字段是用于定义如何处理关联文档的元数据，包括文档的_index、_type、_id 和_source 字段。

3.1.2 修改模式

可以使用 PUT Mapping API 将新类型添加到现有的索引中，或将新字段添加到现有类型中。

首先创建一个没有任何类型映射，叫做 blog 的索引。

```
#curl -XPUT 'localhost:9200/blog?pretty' -H 'Content-Type:
application/json' -d'{}'
```

然后添加名为 user 的新 Mapping 类型。

```
#curl -XPUT 'localhost:9200/blog/_mapping/user?pretty' -H 'Content-Type:
application/json' -d'
{
  "properties": {
    "name": {
      "type": "text"
    }
  }
}'
```

向 user 类型中添加一个名为 email 的新字段。

```
#curl -XPUT 'localhost:9200/blog/_mapping/user?pretty' -H 'Content-Type:
application/json' -d'
{
  "properties": {
    "email": {   //增加一个关键词类型的 email 列
      "type": "keyword"
    }
  }
}'
```

不可能删除某个类型的映射，因为 Elasticsearch 没有提供这样的 API。为了实现这样的效果，可以删除索引，并用新的 Mapping 重新创建索引，如删除 test 索引：

```
# curl -XDELETE localhost:9200/test
```

Elasticsearch 的 Mapping 一旦创建，只能增加字段，而不能修改已有的 Mapping 字段，因此可以创建一个新的索引。但重建索引过程需要更新应用程序以使用新的索引名称，这时，可以使用索引别名来实现修改索引的零停机时间。

索引别名就像一个快捷方式或符号链接，它可以指向一个或多个索引，并且可以在任何需要索引名称的 API 中使用。别名为我们提供了灵活性。例如，为 test 索引创建别名：

```
#curl -XPOST 'http://localhost:9200/_aliases' -d '
   {
      "actions": [
         {"add": {"index": "test", "alias": "alias1"}}
                                          //为 test 增加一个叫做 alias1 的别名
      ]
   }'
```

可以使用 Reindex API 把源索引的数据导入到目标索引中。

Reindex 不会自动建立目标索引，也不会复制源索引的设置，要在运行_reindex 操作之前建立目标索引，包括建立映射、分片计数和副本等。

3.2　Mapping 数据类型

不同的数据类型可以进行不同的操作。例如，对于 Date 型，可以使用“2011 TO 2017”的方式进行范围检索。Elasticsearch 支持的基础数据类型主要有以下几种。

- 字符串类型：string；
- 数值类型：字节（byte）、2 字节（short）、整型（integer）、长整型（long）、浮点型（float）和双精度型（double）；
- 布尔类型：boolean，值是 true 或 false；
- 时间/日期类型：date，用于存储日期和时间；
- 二进制类型：binary；
- IP 地址类型：IP，以字符串形式存储 IPv4 地址；
- 特殊数据类型：token_count，用于存储索引的字数信息。

Binary 类型可以看做是 Base64 编码形式的字符串。例如，存储 Base64 编码的图像字符串，定义 Mapping 代码如下：

```
#curl -XPUT 'localhost:9200/my_index?pretty' -H 'Content-Type:
application/json' -d'
{
  "mappings": {
    "my_type": {                          //定义索引类型"my_type"
      "properties": {
        "name": {                         //定义 text 类型的 name 列
          "type": "text"
```

```
        },
        "img": {                                    //定义 binary 类型的 img 列
          "type": "binary"
        }
      }
    }
  }
}
'
```

这里定义了两列 name 和 img，放入数据：

```
curl -XPUT 'localhost:9200/my_index/my_type/1?pretty' -H 'Content-Type:
application/json' -d'
{
  "name": "Some binary blob",                   //向 name 列放入数据
  "img": "U29tZSBiaW5hcnkgYmxvYg=="             //向 img 列放入数据
}
'
```

Binary 类型默认是只存储，不可查询。其可接受的参数如下。

- doc_values：是否被存储在磁盘，以参加后续的排序、聚合等，默认为 true；
- store：是否存储到_source，并能从_source 检索，默认 false。

字符串类型可以按词建立全文索引。我们可以选择使用切分字符串所用的分词器 analyzer 来实现不同的切分效果，一般不同的语言使用不同的分析器。

Elasticsearch 中内置了很多分词器（analyzers），如 standard（标准分词器）、english（英文分词）和 chinese（中文分词）。其中，standard 分词器对于中文可切分成单个汉字，所以适用范围广，但是搜索准确度低；english 分词器对英文更加智能，可以识别单数、复数、大小写和 stopwords（如 the 这个词）等；chinese 分词器处理中文效果较差。

例如，定义一个使用 english 分词器类型的字段：

```
{
    "blog": {
        "type":     "string",
        "analyzer": "english"
    }
}
```

3.3 Mapping 参数

除了 Mapping 中定义的索引结构信息，还可以在 settings 中设置作用于 index 的一些相关配置信息，如分片数、副本数、tranlog 同步条件和 refresh 时间间隔等。

例如，通过 API 设置，把 refresh 时间设置为 10 秒：

```
curl -XPUT 'localhost:9200/blog/_settings?pretty' -H 'Content-Type:
application/json' -d'
{
```

```
    "index" : {
        "refresh_interval" : "10s"          //设置刷新时间
    }
}
```

当大规模地创建索引操作的时候，最好关闭刷新（refresh），即不重新打开索引。

```
{"refresh_interval": "-1"}
```

可以指定一些字段的 store 属性为 true，这意味着可以单独存储这些数据。这时候，如果你要求返回 field1（store：yes），Elasticsearch 会分辨出 field1 已经被存储了，因此不会从_source 中加载，而是从 field1 的存储块中加载。示例如下：

```
#curl -XPUT 'http://localhost:9200/test/_mapping/t2?pretty' -d '
{
    "_source": {                            //不存储原始内容
        "enabled": false
    },
    "properties": {
        "content": {                        //存储字符串类型的内容列
            "type": "string",
            "store": "true"
        },
        "name": {                           //存储字符串类型的名字列
            "type": "string",
            "store": "false"
        }
    }
}
```

其中，_source 字段存储的是索引的原始内容。

当将一个列的 store 属性设置为 true 时，就会在 Lucene 层面处理存储。Lucene 是倒排索引，可以快速执行全文检索，并返回符合检索条件的文档 ID 列表。除了全文索引，Lucene 也提供了存储字段的值的特性，以支持根据文档 ID 得到原始信息。通常我们在 Lucene 层面存储的列值是跟随 search 请求一起返回的（ID+field 的值）。

有些情况下，显式地存储某些列的值是必须的，比如当_source 被关闭的时候，可能并不想从 source 中解析来得到列的值（即使这个过程是自动的）。请记住，从每一个存储的列中获取值时都需要一次磁盘 I/O，如果想获取多个列的值，就需要多次磁盘 I/O。但是如果要从_source 中获取多个列的值，则只需要一次磁盘 I/O，因为_source 只是一个字段，所以在大多数情况下，从_source 中获取多个列的值是快速而高效的。

3.4　动态 Mapping

在常规情况下，可以不用事先声明 Mapping 而直接放入数据。那么 Elasticsearch 是怎么知道字段是什么类型呢？实际上它是通过给定 document 的 JSON 来判定的。例如，字符串类型的值是用引号引起来的，布尔类型的值是 true 或者 false 等，数字则直接给出。

3.4.1 使用动态 Mapping

执行以下命令：

```
#curl -XPUT http://localhost:9200/test/item/1 -d '{"name":"mick",
"description": "A Pretty cool guy."}'
```

Elasticsearch 能根据值识别出 name 和 description 字段的类型是 string，Elasticsearch 默认会创建以下的 Mapping。

```
mappings: {
    item: {
        properties: {                          //item 索引类型的模式定义
            description: {                     //字符串类型的描述列
                type: string
            }
            name: {                            //字符串类型的名字列
                type: string
            }
        }
    }
}
```

这里使用了 HTTP 协议的 Put 方法用来更新数据。Put 方法和 Get 方法一样，都是幂等的。

Web 资源或 API 方法的幂等性是指一次和多次请求某一个资源应该具有同样的副作用。幂等性是系统接口对外的一种承诺（而不是实现），承诺只要成功调用接口，则外部多次调用对系统的影响是一致的。幂等性是分布式系统设计中的一个重要概念，对超时处理、系统恢复等具有重要意义。声明为幂等的接口会认为外部调用失败是常态，并且失败之后必然会有重试。例如，在因网络中断等原因导致请求方未能收到请求返回值的情况下，如果该资源具备幂等性，请求方只需要重新请求即可，无须担心重复调用会产生错误。Post 方法用于创建资源，每次请求都会产生新的资源，因此不具备幂等性。

3.4.2 实现原理

可以使用有限状态自动机检测数据的类型。例如，匹配数字串的有限状态机：

```
//多次匹配 0～9 之间的数字
Automaton num = BasicAutomata.makeCharRange('0', '9').repeat(1);
num.determinize();                         //转换成确定自动机
num.minimize();                            //最小化

String s = "10899";                        //输入串
boolean accept = num.run(s);               // 判断有限状态机是否接收输入字符串
System.out.println(accept);                //输出 true
```

判断数字类型的完整写法如下：

```
public static Automaton getNum() {
Automaton a = BasicAutomata.makeCharRange('0', '9');
Automaton b = a.repeat(1);          //至少重复一次
//支持 112,345 这样的“三位分节制”记数法
Automaton comma = BasicAutomata.makeChar(',');
Automaton end = BasicOperations.concatenate(comma, a.repeat(1));
Automaton intNum = BasicOperations.concatenate(b, end.repeat());
                                    //数字后接逗号
Automaton comma2 = BasicAutomata.makeChar('.');
Automaton floatNum = BasicOperations.concatenate(comma2, a.
repeat(1));
Automaton intWithFloat = BasicOperations.concatenate(intNum,
        floatNum.optional());       //带浮点数的写法

Automaton percent = BasicAutomata.makeChar('%');
Automaton floatWithPercent = BasicOperations.concatenate
(intWithFloat,
        percent.optional());        //带百分号的写法

//合并百分号的写法与浮点数的写法
Automaton num = BasicOperations.union(floatWithPercent, floatNum);
//确定化自动机
num.determinize();
return num;
}
```

识别英文日期，例如“Nov. 29”这样写法的自动机如下：

```
public static Automaton getDate(){
//月份
Automaton mon = BasicAutomata.makeString("Jan. ");
mon = mon.union(BasicAutomata.makeString("Feb. "));
mon = mon.union(BasicAutomata.makeString("Mar. "));
mon = mon.union(BasicAutomata.makeString("Apr. "));
mon = mon.union(BasicAutomata.makeString("Jun. "));
mon = mon.union(BasicAutomata.makeString("Jul. "));
mon = mon.union(BasicAutomata.makeString("Aug. "));
mon = mon.union(BasicAutomata.makeString("Sept. "));
mon = mon.union(BasicAutomata.makeString("Sep. "));
mon = mon.union(BasicAutomata.makeString("Oct. "));
mon = mon.union(BasicAutomata.makeString("Nov. "));
mon = mon.union(BasicAutomata.makeString("Dec. "));

//1 到 2 位数字表示的日期
Automaton num = BasicAutomata.makeCharRange('0', '9').repeat(1,2);

//日期后缀
Automaton suffix = BasicAutomata.makeString("rd");
suffix = suffix.union(BasicAutomata.makeString("th"));
suffix = suffix.union(BasicAutomata.makeString("st"));
suffix = suffix.union(BasicAutomata.makeString("nd"));
```

```
//日期后缀是可选的
Automaton date = mon.concatenate(num).concatenate(suffix.optional());
date.determinize();
return date;
}
```

3.5 本章小结

Mapping 是对索引库中索引的字段名称及其数据类型进行定义，类似于数据库中的表结构信息。但是 Elasticsearch 的 Mapping 比数据库灵活得多，它可以动态识别字段，字符串映射为 string，数字映射为 long。如果需要对某些字段添加特殊属性如定义使用其他分词器、是否分词、是否存储等，就必须手动添加 Mapping。

第 4 章　深入源码分析

本章首先介绍 Elasticsearch 使用的 Lucene 源码并进行分析，然后介绍 Elasticsearch 本身的源代码以及它所使用的网络通信框架 Netty。

4.1　Lucene 源码分析

本节首先介绍如何使用 Lucene API，然后介绍编译 Lucene 源码所要用到的软件工具 Ivy，最后分析 Lucene 源码实现过程。

4.1.1　使用 Lucene

Lucene 完成基本的搜索功能只需要一个不依赖外部程序包的 JAR 文件，因为该文件是一个核心文件，所以叫做 lucene-core-Version.jar。例如，Lucene 的 7.1.0 版本叫做 lucene-core-7.1.0.jar，可以从 http://lucene.apache.org/core/网站上下载这个 jar 包。

Lucene 把待查询的文档集合按词组织成倒排索引，其中的索引库叫做 Index，是位于一个目录下的一些二进制文件。和一般的数据库不一样，Lucene 不支持定义主键，在 Lucene 中并不存在一个叫做 Index 的类。Lucene 中通过 IndexWriter 写索引，通过 IndexReader 读索引。

我们首先介绍如何创建索引库，然后介绍如何搜索索引库。总的来说，往 Lucene 中放的是文档，查询的是词，查询返回的也是文档。使用 Lucene 实现搜索的基本流程如图 4-1 所示。

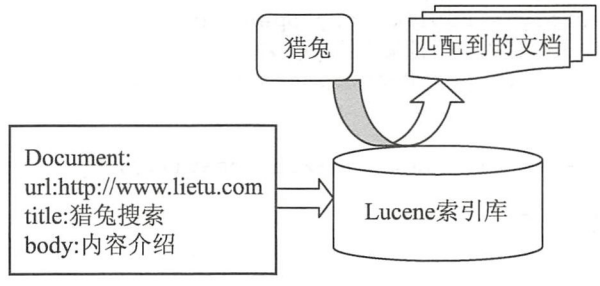


图 4-1　Lucene 搜索的基本流程示意图

我们在 Eclipse 中创建一个 Java 控制台项目，创建 lib 目录，然后把 lucene-core-7.1.0.jar 文件复制到 lib 目录下。在项目属性中增加对 lucene-core-7.1.0.jar 文件的引用。

创建索引时需要指定切分文本用的分析器，这里使用 StandardAnalyzer 分析器切分文本。因为 StandardAnalyzer 分析器位于 lucene-analyzers-common-7.1.0.jar 文件中，所以需要增加对这个文件的引用。把 lucene-analyzers-common-7.1.0.jar 文件复制到 lib 目录下，然后在项目属性中增加对 lucene-analyzers-common-7.1.0.jar 文件的引用。

新建一个测试类，实现在硬盘中创建索引：

```
//创建 StandardAnalyzer
Analyzer analyzer = new StandardAnalyzer();
// 把索引存在硬盘上的一个目录中
Path path = Paths.get("d:/news");
Directory directory = FSDirectory.open(path);          //存放新闻的索引
IndexWriterConfig config = new IndexWriterConfig(analyzer);
IndexWriter iwriter = new IndexWriter(directory, config);
                                                       //IndexWriter 写索引
Document doc = new Document();
String text = "This is the text to be indexed.";   //要索引的文本
doc.add(new Field("title", text, TextField.TYPE_STORED));
iwriter.addDocument(doc);                              //增加文档
iwriter.close();
directory.close();
```

查询串中可能包括一些高级查询语法。例如，要找包含 Java 的 PDF 文件，可以使用查询串 java filetype:pdf。所以用查询分析器 QueryParser 来解析查询串，也就是根据查询串生成 Query 对象。QueryParser 这个类位于 org.apache.lucene.queryparser.classic 包中，需要引用 lucene-queryparser-7.1.0.jar 文件。下面新建一个测试类来查询索引：

```
String defaultField = "title";
String queryString = "test";
Analyzer analyzer = new StandardAnalyzer();
QueryParser parser = new QueryParser(defaultField, analyzer);
                                                       // 用于解析查询语法
// 从字符串得到查询对象
Query query = parser.parse(queryString);
// 存放新闻的索引路径
Path path = Paths.get("d:/news");
// 把索引存在硬盘上的一个目录中
Directory directory = FSDirectory.open(path);
// DirectoryReader 读入一个目录下的索引文件
IndexReader ir = DirectoryReader.open(directory);

// 打开索引库
IndexSearcher searcher = new IndexSearcher(ir);

// 根据查询词搜索索引库
TopDocs hits = searcher.search(query, 10);             // 最多返回 10 个结果
System.out.println("hits.totalHits:" + hits.totalHits);
for (int j = 0; j < hits.scoreDocs.length; j++) {
```

```
        // 根据文档编号取出文档对象
        Document hitDoc = searcher.doc(hits.scoreDocs[j].doc);
        System.out.println(hitDoc.get("title"));        // 输出文档
    }
```

4.1.2　Ivy 管理依赖项

可以使用 git 命令从 github.com 得到 Lucene 最新的源码。

```
#git clone https://github.com/apache/lucene-solr.git
```

Lucene 源代码采用 Ant 构建，使用 Apache Ivy（下载地址是 http://ant.apache.org/ivy/）管理 JAR 文件之间的依赖关系。

Ivy 特有的文件是 ivy.xml 和一个 Ivy 设置文件，ivy.xml 文件中列举了项目的所有依赖项。例如，nutch 依赖 HttpClient：

```
<dependency org="commons-httpclient" name="commons-httpclient"
      rev="3.1" conf="*->master" />
```

Ivy 依赖于 Ant，所以需要先安装 Ant，然后下载 Ivy，再将它的 JAR 文件复制到 Ant 的 lib 下，这样就可以在 Ant 里使用 Ivy 进行依赖管理了。

首先将 Apache Lucene-Solr 导入 Eclipse 中：

```
ant compile
ant eclpise
```

然后就可以选择菜单 File | Import | Existing Projects Into workspace 命令，导入 Eclipse 了。

4.1.3　源码结构介绍

Lucene 源码分为核心包和外围功能包。核心包实现搜索功能，外围功能包实现高亮显示等辅助功能。Lucene 源码的核心包中包括 7 个基本的功能子包，每个包完成特定的功能。

Lucene 源码核心包中最基本的是索引管理包（org.apache.lucene.index）和检索管理包（org.apache.lucene.search）。索引管理包实现索引建立、删除等功能；检索管理包根据查询条件，检索得到结果。

索引管理包调用数据存储管理包（org.apache.lucene.store），主要包括一些底层的 I/O 操作，同时它也会调用一些公用的算法类（org.apache.lucene.util）。编码管理包（org.apache.lucene.codecs）用于方便自定义索引的编码和结构；文档结构包（org.apache.lucene.document）用于描述索引存储时的文档结构管理，类似于关系型数据库的表结构。

查询分析器包（org.apache.lucene.queryParser）实现查询语法，支持关键词间的运算，如与、或、非等。语言分析器（org.apache.lucene.analysis）主要用于对放入索引的文档和查询词进行切词，支持中文扩展此类。

索引一般以文件形式存储在磁盘上，索引检索需要磁盘 I/O 操作。与主存不同，磁盘

I/O 存在机械运动耗费的特点，因此磁盘 I/O 的时间消耗是巨大的。由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高 I/O 效率。

预读的长度一般为页（Page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页的大小通常为 4KB），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页数据载入内存中，然后返回异常，程序继续运行。

在 Lucene 源代码中，为了和搜索结果页中的翻页区分，缓存单元不叫 Page，而叫做 Block。

为了方便索引大量的文档，Lucene 中的一个索引分成了若干个子索引，这个子索引叫做段（segment），段中包含了一些可搜索的文档。在给定的段中可以快速遍历任何给定索引词在所有文档中出现的频率和位置。IndexWriter 收集在内存中的多个文档，然后在某个时间点把这些文档写入一个新的段，写入点可以通过 Lucene 内部的配置类或者外部程序控制。然后这些文档组成的段会保持不动，直到 Lucene 把它合并入大的段中。MergePolicy 控制 Lucene 如何合并段，如图 4-2 所示。

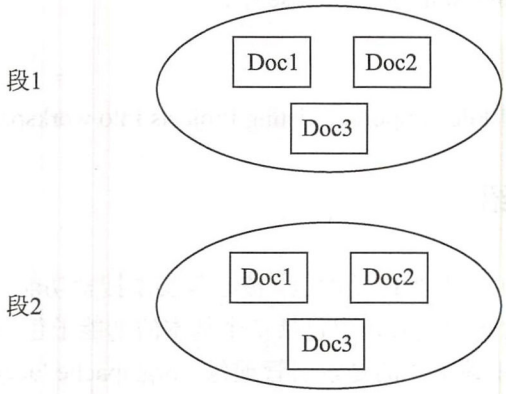


图 4-2　索引文件的逻辑视图

SegmentCoreReader 关联整个段中的所有文件，通过（编码器）得到各个文件的处理对象。

Codec（编译码器）事实上就是由多组的 format（格式）构成，一个 Codec 总共包含 8 个 format，分别是 PostingsFormat、DocValuesFormat、StoredFieldsFormat、TermVectorsFormat、FieldInfosFormat、SegmentInfoFormat、NormsFormat 和 LiveDocsFormat。例如，StoredFields Format 用来处理存储数据的列，TermVectorsFormat 用来处理词向量。

Codec 直接传递给 SegmentReader（段读取器）来编码索引格式；提供枚举类的实现给 SegmentReader；提供索引文件的写入器给 IndexWriter。

PerFieldCodec 用来支持不同的列使用不同的读写格式。org.apache.lucene.codecs.perfield 包中有可以委托给每个字段不同格式的 Posting 格式。

和其他的 Codec 写入到压缩的二进制文件不一样，SimpleTextCodec 把所有的投递列表写到可读的文本文件中，其适合用来学习，不建议在生产环境中使用。代码如下：

为了方便测试查询，我们使用 createDocument()方法索引测试内容。

```
private static Document createDocument(String id, String content) {
                                                    //创建文档
Document doc = new Document();
doc.add(new Field("id", id, StringField.TYPE_STORED));
                                                    //索引不分词的字符串列
//索引分词的文本列
doc.add(new Field("contents", content, TextField.TYPE_STORED));
return doc;
}
```

使用 SimpleTextCodec 建立索引：

```
Analyzer analyzer = new StandardAnalyzer();
IndexWriterConfig iwc = new IndexWriterConfig(analyzer);

iwc.setCodec(new SimpleTextCodec());        //设置编码器为 SimpleTextCodec
iwc.setUseCompoundFile(false);                  //索引不使用复合文件格式
Path path = Paths.get("F:/lucene/index");       //索引存放路径
Directory directory = FSDirectory.open(path); //打开这个路径

IndexWriter writer = new IndexWriter(directory, iwc);
                                                    //构建写索引的 IndexWriter
// 索引一些文档
writer.addDocument(createDocument("1", "青菜鸡肉"));
writer.addDocument(createDocument("2", "老鸭粉丝汤"));
writer.addDocument(createDocument("3", "辣子鸡丁"));
writer.close();
```

主要产生 5 个文件：_0.fld、_0.len、_0.inf、_0.pst 和_0.si。其中：

- fld 文件保存存储到索引的原值；
- pst 文件保存倒排索引；
- inf 文件保存文件是如何索引的。

存储原值的_0.fld 文件内容如下：

```
doc 0                                           //文档 0
  numfields 2                                   //列的数量
  field 0                                       //第 0 列
    name id                                     //列名称
    type string                                 //列类型
    value 1                                     //值
  field 1                                       //第 1 列
    name contents                               //列名称
    type string                                 //列类型
```

```
    value 青菜鸡肉                    //值
doc 1                                 //文档 1
  numfields 2                         //列的数量
  field 0                             //第 0 列
    name id                           //列名称
    type string                       //列类型
    value 2                           //值
  field 1                             //第 1 列
    name contents                     //列名称
    type string                       //列类型
    value 老鸭粉丝汤                  //值
doc 2                                 //文档 2
  numfields 2                         //列的数量
  field 0                             //第 0 列
    name id                           //列名称
    type string                       //列类型
    value 3                           //值
  field 1                             //第 1 列
    name contents
    type string
    value 辣子鸡丁
END
```

倒排索引在_0.pst 文件中，先保存某一列的倒排索引，然后再保存另外一列的倒排索引。例如，像这样写入 contents 列和 id 列的倒排索引：

```
field contents                        //contents 列的倒排索引
  term 丁                             //词
    doc 2                             //文档编号
      freq 1                          //词在文档中出现的次数
      pos 3                           //词出现的位置
  term 丝
    doc 1
      freq 1
      pos 3
  term 子
    doc 2
      freq 1
      pos 1
  term 汤
    doc 1
      freq 1
      pos 4
  term 粉
    doc 1
      freq 1
      pos 2
  term 老
    doc 1
      freq 1
      pos 0
```

```
  term 肉
    doc 0
      freq 1
      pos 3
  term 菜
    doc 0
      freq 1
      pos 1
  term 辣
    doc 2
      freq 1
      pos 0
  term 青
    doc 0
      freq 1
      pos 0
  term 鸡
    doc 0
      freq 1
      pos 2
    doc 2
      freq 1
      pos 2
  term 鸭
    doc 1
      freq 1
      pos 1
field id                                          //id 列的倒排索引
  term 1
    doc 0
  term 2
    doc 1
  term 3
    doc 2
END
```

_0.inf 文件存储索引的元信息内容如下：

```
number of fields 2                                //列数量
  name id                                         //id 列的说明
  number 0                                        //列编号
  indexed true                                    //索引
  index options DOCS_ONLY                         //索引选项
  term vectors false                              //词向量
  payloads false                                  //载荷
  norms false                                     //归一化
  norms type false                                //归一化类型
  doc values false                                //文档值
  attributes 0                                    //属性
  name contents                                   //内容列的说明
  number 1
  indexed true
  index options DOCS_AND_FREQS_AND_POSITIONS
```

```
term vectors false
payloads false
norms true
norms type NUMERIC
doc values false
attributes 0
```

fld 文件包含一个叫做 checksum 的二进制序列，用来检验文件的完整性，该二进制序列采用 CRC32 算法生成。

4.1.4 并发控制

索引库为了实现读写并发控制，需要对正在读取索引的应用进行引用计数，当没有任何应用读取索引时，也就是引用计数为 0 后，可以提交修改。引用计数需要保持连续性，这时候需要生成一个全局唯一的序列号。例如，最容易想到的一个实现方法是：

```
public class UnsafeSequence {
    private int value;
    /** 返回一个唯一的值 */
    public int getNext() {
        return value++;
    }
}
```

如果一个线程调用这个方法，则不会有问题。但如果多个线程调用这个方法，则会出现问题。下面来测试两个线程。

```
public class TestUnsafeSequence extends Thread {
       static UnsafeSequence unsafeSequence = new UnsafeSequence();
                                                    //序列号生成器
     static HashSet<Integer> seqSet = new HashSet<Integer>();
                                                //已经生成的

    public void run() {
     while (true){
         int id = unsafeSequence.getNext();
         if(seqSet.contains(id)){
             System.out.println("序列号重复错误: "+id);
         }
         seqSet.add(id);
    }
}

    public static void main(String args[]) {
        (new TestUnsafeSequence()).start();
        (new TestUnsafeSequence()).start();
    }
}
```

输出结果如下：

```
序列号重复错误：6503
序列号重复错误：7582
序列号重复错误：7849
序列号重复错误：7971
序列号重复错误：12599
序列号重复错误：13175
...
```

返回值重复是因为访问的是同一个 value 值，先取得这个值，然后加 1。自增操作 value++看起来像是一个单一的操作，但是事实上分为 3 个独立的执行操作：读取这个值，使之加 1，再写入新值。因为这些操作发生在多个线程中，这些线程可能交替占有运行时间，所以两个线程很可能同时读取这个值，而且这两个线程都得到了相同的值，并都增加了 1，结果就是不同的线程返回了相同的序列数。如图 4-3 所示为运气不好的一次执行过程。

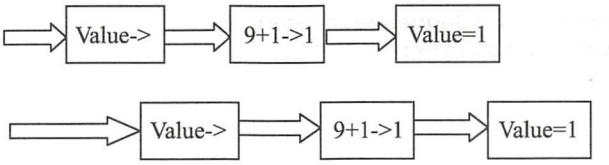


图 4-3　运气不好的一次执行过程

判断 value 的值叫做竞争条件。避免竞争条件有以下 3 个方法。

- 无共享：如果能够互不相干地使用，则没问题。例如，数据按类别交给不同的线程去处理。
- 使用原子操作：所有对 value 值的操作一次性完成，中间不可中断。
- 使用锁：对需要同步的整个方法加锁。

AtomicInteger 中的 incrementAndGet()方法是原子性的，这个类位于 java.util.concurrent.atomic 包中，可以用它实现序列号生成器。

```
public class SafeSequence{
  private final AtomicInteger value = new AtomicInteger (0);

  public int getNext() {
    return value.incrementAndGet();          //原子性的操作
  }
}
```

我们可以用同样的方法测试 SafeSequence 中的序列号生成器。为了避免测试中重复生成序列号的情况，可以把 getNext 声明为 synchronized 类型的方法来修正 UnsafeSequence，这样就能避免图 4-3 所示的那种不应出现的交互。

```
public class Sequence {
  private int value;

  public synchronized int getNext() {
```

```
        return value++;
    }
}
```

synchronized getNext(){}可以防止多个线程同时访问这个对象的 getNext 方法。同时，如果一个对象有多个 synchronized()方法，只要一个线程访问了其中的一个 synchronized()方法，那么其他线程就不能同时访问这个对象中的任何一个 synchronized()方法，也就是说 synchronized()方法是对象级的锁。这时，不同的对象实例的 synchronized()方法是不相干扰的，也就是说，其他线程照样可以同时访问相同类的另一个对象实例中的 synchronized()方法。

除了对象级的锁，还有类级别的锁。例如：

```
public class Sequence {
        private static int value;

        private static int getNext() {
         synchronized (Sequence.class) {
             return value++;
         }
    }
}
```

这里的 synchronized()方法锁的是一个类，Sequence.class 本身是 Sequence 类的一个静态属性，也是一个对象。锁 Sequence.class 的意思就是对整个类加锁，也就是说无论创建了多少个 Sequence 类的对象，这些对象都共享一个相同的锁标记。

上面的例子只锁了一个变量，对于每个不止涉及一个变量的不变式，不变式中所有的变量必须都由一个锁保护。

对于高并发应用，最好使用 ConcurrentHashMap，而不是 Hashtable。Hashtable 虽然是同步的，但是实际上往往需要“检查然后放入”这样的原子操作，例如下面的代码：

```
HashMap<String,Integer> myMap = new HashMap<String,Integer>();
Collections.synchronizedMap(myMap);
synchronized(myMap) {
   if (!myMap.containsKey("tomato"))        //检查不存在
      myMap.put("tomato", 1);               //放入
}
```

所以在一般情况下不推荐使用 Hashtable。当执行任务需要较长时间时，不应该使用锁，例如 I/O 操作等。

访问主板上的内存速度远不及 CPU 处理速度。为提高机器整体性能，在 CPU 内部引入高速缓存，加速对内存的访问。有的 CPU 内部共有三级缓存，分别是 L1（一级缓存）和 L2（二级缓存）及 L3（三级缓存）。

同一个花生植株可以结多个花生果，一个花生果中有多粒花生。类似的，一台机器可以有多个 CPU，一个 CPU 可以有多个核心。在 x86 和 x64 中，处理器设计成同步不同处理器中的高速缓存，所以我们可能看不出问题。但 IA64 处理器利用了这个特点：每个处理器都有其自己的高速缓存，各处理器中的缓存数据不严格同步。所以，不同的线程执行

可能在缓存中放进不同的值。

因此，在第一次运行时，CPU 访问内存地址并把值存储在缓存中，当第二次访问变量时就从缓存中返回，所以所有后续读取都从缓存读取。写操作也是同样的，当变量改变时，写入值首先存储在缓存中，随后的读/写也是从缓存中读/写。然而，当写入值最终被刷新到内存中时，则清除缓存或用其他数据填充缓存。CPU 比较“聪明”地做了一件事：当 CPU 在缓存中获取变量的值（几个字节）时，同时也获取了该值附近的一些值，因为下一个要使用的变量可能就在它附近。具体的缓存值获取过程如图 4-4 至图 4-7 所示。

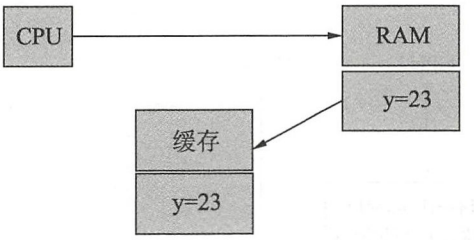


图 4-4　第一次获取

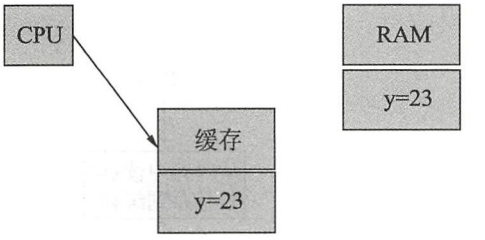


图 4-5　第二次获取

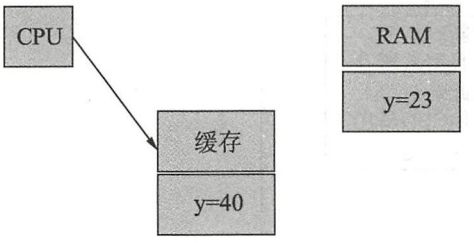


图 4-6　改变值

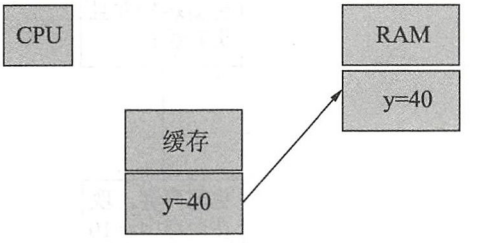


图 4-7　未来的某个时候

一个 CPU 可以写它自己的缓存，这最终会转移到 RAM，其他 CPU 可能已经从 RAM 中读过这个 CPU 尚未更新的值，如图 4-8 所示。

因为缓存，多个 CPU 访问同一个变量时导致能见度降低。在一个线程中发生的内存写有可能被另外一个变量看到。也就是说一个线程写了一个变量后，并不能保证能被另外一个线程看到。因此这就产生了明显的并发问题。这样当多个线程同时与某个对象交互时，就必须要让线程及时地得到共享成员变量的变化，这是产生 volatile 关键字的原因。

如果你声明一个 volatile 变量，会总是从内存中读取它，并把它的值立即写入内存中。但是必须指出的是，所有的锁操作都会同步缓存。因此，在锁操作内部并不需要 volatile 关键字。

总之，对于 volatile 关键字修饰的成员变量，不能保存它的私有备份，而应直接与共享成员变量交互。

例如，下面是一个停止请求的方法，允许其他线程通知这个线程结束任务。

```
public class StoppableTask extends Thread {
```

```
    private volatile boolean pleaseStop;

    public void run() {
      while (!pleaseStop) {
        // 做一些事情...
      }
    }

    public void tellMeToStop() {          //其他线程调用这个方法让这个线程停止
      pleaseStop = true;
    }
}
```

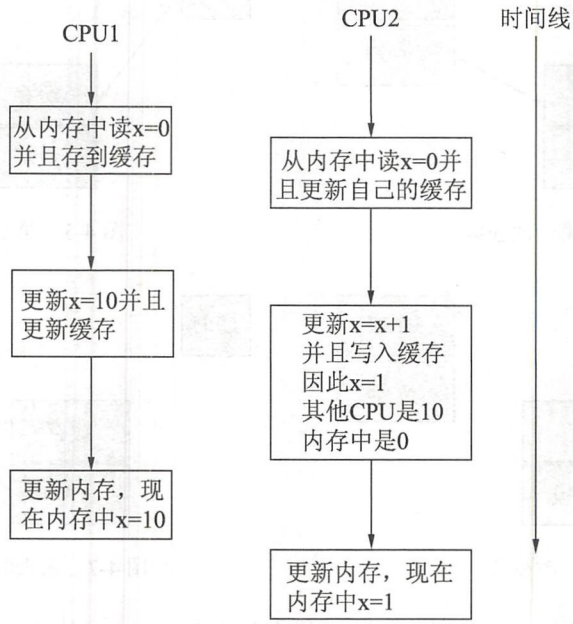


图 4-8　两个 CPU 访问同一个变量

如果该变量没有声明为 volatile（并且也没有其他同步措施），那么这是合法的，运行循环的线程在循环开始时缓存变量 pleaseStop 的值，并且不再看它。如果不希望这是一个无限循环，就需要使用 volatile 修饰 pleaseStop 变量，调用停止任务的方法。代码如下：

```
public static void main(String[] args) throws InterruptedException {
    StoppableTask task = new StoppableTask();
    task.start();
    Thread.sleep(3000);                    //3 秒后停止任务
    task.tellMeToStop();
}
```

例如，如果想让宾馆的服务员打扫房间，客人可以在房门外设置一个请打扫的标志；如果不希望打扫房间，也可以在房门外设置一个请勿打扰的标志。服务员和客人通过一个位置的标志来交流信息。这里的代码中使用 volatile 修饰的 pleaseStop 变量作为交流信息

的标志。

ready 是一个 volatile 布尔变量，初始值是 false，而 answer 是一个非 volatile 的整数变量，初始值是 0。

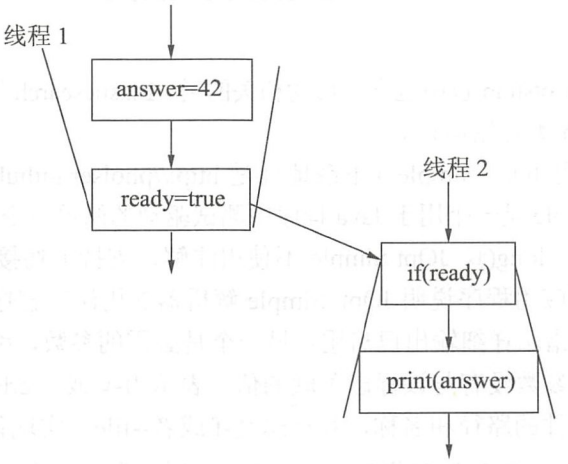


图 4-9　两个线程的梯形图

第一个线程写入 ready，这是通信的发送方；第二个线程读取 ready，并打印出它看到的第一个线程的值，因此它是一个接收器。在这两个线程通信时，线程 1 在设置 ready 为 true 之前，可以看到所有的内存内容，在线程 2 读出 ready 的值为 true 之后，必须让线程 2 可以看到所有的内存内容。

每次对一个 volatile 变量的写入就是一个同步点，每个同步点都会有潜在的性能损失。不把多个变量都声明成为 volatile 变量，是为了减少性能损失。

为了提高执行速度，CPU 中指令的执行并不一定严格按照顺序来执行，如果没有相关性的指令，可以乱序执行，以充分利用 CPU 的指令流水线提高执行速度。这是硬件级别的优化。

编译器优化代码常用的方法有：将内存变量缓存到寄存器中；调整指令顺序充分利用 CPU 指令流水线，常见的是重新排序读写指令。对常规内存进行优化的时候，这些优化是透明的，而且效率很好。由编译器优化或者硬件重新排序引起的问题解决办法是，从硬件（或者其他处理器）的角度看，对必须以特定顺序执行的操作设置内存屏障（memory barrier）。

```
volatile int [] arr = new int[SIZE];

arr = arr;
int x = arr[0];
arr[0] = 1;
```

arr 是对数组的 volatile 类型引用，不是对 volatile 元素组成的数组引用，因此写入 arr[0] 不是对 volatile 类型的写入。如果写入 arr[0]，则不会得到像写入 volatile 类型一样的顺序，

并且多个 CPU 都不能正常访问这个 arr[0]变量。

4.2 启动搜索服务

org.elasticsearch.bootstrap 包中包含了启动相关的类。Elasticsearch 服务从 org.elasticsearch.bootstrap.Elasticsearch 类开始运行。

Elasticsearch 使用 JOpt Simple（下载地址是 http://pholser.github.io/jopt-simple/）解析命令行参数。JOpt Simple 是一个用于 Java 程序中测试驱动的简单命令行解析器，支持 POSIX getopt()和 GNU getopt_long()。JOpt Simple 不使用注解，支持流畅接口。

下面用一个虚拟应用程序说明 JOpt Simple 解析器的用法。程序中有两个参数，一个是可选的参数，用来指示详细输出已启用；另一个是必需的参数，用于指定虚拟应用程序要处理的文件。可选参数没有与该标志关联的值，表示为-v 或--verbose；必需的参数后面应该跟一个值，即文件的路径和名称，其标志是-f 或者--file。实现代码如下：

```
final OptionParser optionParser = new OptionParser();

//文件参数
final String[] fileOptions = {
    "f",
    "file"
};

//带参数，而且是必需
optionParser.acceptsAll(Arrays.asList(fileOptions),
      "Path and name of file.").withRequiredArg().required();

//详细输出参数
final String[] verboseOptions = {
    "v",
    "verbose"
};

optionParser.acceptsAll(Arrays.asList(verboseOptions),
       "Verbose logging.");

//帮助参数
final String[] helpOptions = {
    "h",
    "help"
};

optionParser.acceptsAll(Arrays.asList(helpOptions),
     "Display help/usage information").forHelp();
```

首先实例化一个 OptionParser，然后为每个可能的命令行选项调用一个重载的 acceptAll()方法。acceptAll()方法允许多个标志/选项名称与单个选项相关联。对选项同义词的这种支

持允许使用-f 和--file 作为相同的选项。

上面的代码演示了可以使用.required()方法调用来指定一个命令行选项是必须的。在这种情况下，file 是必需的。如果预期将参数放置在与选项/标志关联的命令行上，则可以使用 withRequiredArg()方法。

上面代码段中的 help 选项利用 forHelp()方法告诉 JOpt Simple 解析器：如果在 forHelp()相关的选项处于选中状态时，必需的选项不在命令行中则不会引发异常。在这个例子中，这样可以确保用户使用-h 或--help 运行应用程序，而不需要其他任何必要的选项，并避免引发异常。

从 OptionParser.accepts(String)方法返回 OptionSpecBuilder 的实例。例如返回版本选项：

```
OptionSpecBuilder versionOption = parser.acceptsAll(Arrays.asList("V",
"version"),
   "Prints elasticsearch version information and exits");
```

4.3　Guice 框架

Elasticsearch 在启动时，基于其配置文件和运行时的环境来搜集不同的模块，并创建一个 Injector 对象。简单来说，Injector 就是一个不需要提供构建参数就可以构建类的实例对象。Injector 将会使用其配置完的模块来定位所有请求的依赖，并以一种拓扑顺序为开发者建出这些实例。这样不仅为开发者节约了大量时间，而且可以帮助开发者创建出一个可复合的模块系统。

Elasticsearch 服务启动时通过 ModuleBuilder 类进行模块注入。ModuleBuilder 是 Guice 的封装。Guice 是 Google 公司开发的一个开源依赖注入框架（IOC）。Elasticsearch 源代码中集成了 Guice。相关代码位于 org.elasticsearch.common.inject 包中。Guice 会扫描 inject 注释，并对方法中出现的参数实例寻找对应注册的实例进行初始化。

Elasticsearch 中的模块是在 Guice 模块部件中完成配置信息并绑定 Elasticsearch 各类接口的特定实现。

Guite 的 Provider 类可以返回特定类型的对象。Elasticsearch 通过 Provider 类创建和返回 Analyzer 对象。

下面是使用 Provider 的例子。首先定义一个接口：

```
public interface MyInterface {
   String foobar();
}
```

然后是接口的实现类 MyClass，如下：

```
public class MyClass implements MyInterface {
    private String providerName;                    //记录提供者的名字

    public MyClass(String providerName) {
```

```
        this.providerName = providerName;
    }

    @Override
    public String foobar() {
    return String.format("Hi! I am [%s], " + "and I was instantiated
    using [%s]", getClass().getSimpleName(), providerName);
                                                    //返回调用信息
    }

}
```

Provider 的子类 MyInterfaceProvider 提供 MyClass 的实例（instance）如下：

```
import com.google.inject.Provider;

public class MyInterfaceProvider implements Provider<MyClass> {
                                                    //实现提供者接口

    @Override
    public MyClass get() {
      return new MyClass(getClass().getSimpleName());
                                                    //用类名实例化 MyClass
    }

}
```

Module 的子类建立绑定，代码如下：

```
import com.google.inject.AbstractModule;

public class MyModule extends AbstractModule {

@Override
protected void configure() {
    //绑定 MyInterface 接口到 Provider 子类
    bind(MyInterface.class).toProvider(MyInterfaceProvider.class);
}

}
```

使用 Guice 得到 MyInterface 的对象：

```
import com.google.inject.Guice;
import com.google.inject.Injector;

public class ProviderSample {

    public static void main(String[] args) {
        Injector injector = Guice.createInjector(new MyModule());
                                                    //创建注入器
        MyInterface myObject = injector.getInstance(MyInterface.class);
                                                    //通过注入器得到实例

        System.out.println(myObject.foobar());      //输出调用信息
}
}
```

运行 ProviderSample 输出：

```
Hi! I am [MyClass], and I was instantiated using [MyInterfaceProvider]
```

4.4　日期和时间库——Joda-Time

Elasticsearch 内部使用 Joda-Time 库（下载地址是 http://www.joda.org/joda-time/）处理日期和时间。Joda-Time 库中提供了 date 和 time 类的替换。库使用 Joda-Time 库解析日期字符串的代码如下：

```
DateTimeFormatterBuilder builder = new DateTimeFormatterBuilder();
                                        //日期格式构建器
DateTimeParser[] parsers = new DateTimeParser[3]; //日期字符串解析器数组
parsers[0] = DateTimeFormat.forPattern("MM/dd/yyyy")
                                        //"MM/dd/yyyy"格式的解析器
    .withZone(DateTimeZone.UTC).getParser();
parsers[1] = DateTimeFormat.forPattern("MM-dd-yyyy")
                                        //"MM-dd-yyyy"格式的解析器
    .withZone(DateTimeZone.UTC).getParser();
parsers[2] = DateTimeFormat.forPattern("yyyy-MM-dd HH:mm:ss")
                                        //"yyyy-MM-dd HH:mm:ss"格式的解析器
    .withZone(DateTimeZone.UTC).getParser();

//使用解析器数组构建日期格式构建器
builder.append(
    DateTimeFormat.forPattern("MM/dd/yyyy")
                .withZone(DateTimeZone.UTC).getPrinter(), parsers);

DateTimeFormatter formatter = builder.toFormatter();
                                        //得到日期格式化对象
long millis = formatter.parseMillis("2009-11-15 14:12:12");
                                        //由日期字符串解析出微秒
System.out.println(millis);
```

Elasticsearch 中的 StrictISODateTimeFormat 类从 Joda 中复制过来的，该类在 Joda-Time 中被命名为 ISODatetimeFormat，但是其在几个方法中被修改很多，如日期年份至少为 *n* 位数；像“5”这样的年份是无效的，必须是“0005”。

例如，创建时间对象的代码如下：

```
MutableDateTime dateTime = new MutableDateTime(3000, 12, 31, 23, 59, 59,
999,
                    DateTimeZone.UTC);
System.out.println(dateTime);
```

4.5　Transport 模块

Transport 模块用于 Elasticsearch 集群内节点之间的内部通信，从一个节点到另一个节

点的调用都通过 Transport 模块来完成。

网络线程中不能调用阻塞代码，所以需要把网络线程和非网络线程区分开。枚举类型 Transports 中的静态方法 isTransportThread()根据线程的名字判断一个线程是否为网络线程。

Elasticsearch 集群内默认节点的连接数量是 13 个，如下：

```
ConnectionProfile profile =
   TcpTransport.buildDefaultConnectionProfile(Settings.EMPTY);
assertEquals(13, profile.getNumConnections());

//节点之间的 ping 连接个数为 1 个
assertEquals(1, profile.getNumConnectionsPerType
(TransportRequestOptions.Type.PING));
//典型的搜索和单 doc 索引，默认个数为 6 个
assertEquals(6, profile.getNumConnectionsPerType
(TransportRequestOptions.Type.REG));
//集群状态的发送，默认个数为 1 个;
assertEquals(1, profile.getNumConnectionsPerType
(TransportRequestOptions.Type.STATE));
//做数据恢复 recovery，默认个数为 2 个
assertEquals(2, profile.getNumConnectionsPerType
(TransportRequestOptions.Type.RECOVERY));
//用于批量请求，默认个数为 3 个;
assertEquals(3,
profile.getNumConnectionsPerType(TransportRequestOptions.Type.BULK));
```

在上面代码中，连接配置 ConnectionProfile 中描述了为每个可用请求类型建立了多少到特定节点的连接。

4.6 线程池

每个 Elasticsearch 节点内部都维护着多个线程池，如 index、search、warmer 和 bulk 等，开发者可以修改线程池的类型和大小。

可以通过如下命令查看线程池情况。

```
# curl http://localhost:9200/_nodes/stats?pretty
```

例如，返回的搜索线程池相关信息如下：

```
"search" : {
        "threads" : 2,
        "queue" : 0,
        "active" : 0,
        "rejected" : 0,
        "largest" : 2,
        "completed" : 2
      },
```

其中，最需要关注的是 rejected。当某个线程池 active 的值等于 threads 时，表示所有

线程都在忙，那么后续新的请求就会进入队列中，一旦队列大小超出限制，那么 Elasticsearch 进程将拒绝请求，相应的拒绝次数就会累加到 rejected 中。

4.7　模块

Elasticsearch 使用模块提供分布式搜索系统所需要的实例和功能，AnalysisModule 是使用 Elasticsearch 的开发者最关心的模块之一，这个模块负责提供索引和搜索时分析器、分词器、字符过滤器和 Token 过滤器。

Elasticsearch 内置的模块介绍如下。

- transport-netty4：用于网络通信；
- aggs-matrix-stats：矩阵聚合在多个字段上工作，并产生一个矩阵作为输出；
- analysis-common：包含一些常用的 TokenFilter；
- ingest-common：实现摄取的公用类；
- lang-expression：实现表达式脚本引擎；
- lang-mustache：Mustache 脚本引擎；
- lang-painless：painless 脚本引擎；
- parent-join：父连接；
- percolator：实现注册查询功能；
- reindex：重新索引 Elasticsearch 数据；
- repository-url：用于 URL 存储的模块。

可以使用参数设定模块，这些模块参数可以静态或者动态地进行设定。必须在节点级别设置这些模块的静态方式。在启动节点时，可以在 elasticsearch.yml 文件中设置参数，或者作为环境变量在命令行上设置参数。这些参数必须在集群中的每个相关节点上设置。也可以使用 cluster-update-settings API 在实时群集上动态更新。例如，设定索引恢复的速度代码如下：

```
curl -XPUT 'localhost:9200/_cluster/settings?pretty' -H 'Content-Type:
application/json' -d'
{
    "persistent" : {
        "indices.recovery.max_bytes_per_sec" : "50mb"
    }
}'
```

4.8　Netty 通信框架

Netty（下载地址是 http://netty.io/）是一个 NIO 客户端服务器框架，Elasticsearch 采用

Netty 实现 HTTP 异步通信协议。

Elasticsearch 中的服务器端 Netty4HttpServerTransport 位于 transport-netty4 模块，客户端的 PreBuiltTransportClient 也依赖 transport-netty4 模块。

Netty 是 Reactor 设计模式的一个实现。Reactor 设计模式是用于处理由一个或多个输入同时发送到服务处理程序的服务请求的事件处理模式。然后，服务处理器多路复用输入的请求并将其同步分派到相关联的请求处理程序中。

ServerBootStrap 负责引导服务器启动 NIO 服务。使用 ServerBootStrap 的代码如下：

```
int workerCount=1;

ServerBootstrap serverBootstrap = new ServerBootstrap();
ThreadFactory f = new DefaultThreadFactory("thread pool"); //守护线程工厂
//Reactor 单线程模型
//I/O 事件作为一个触发器，网络请求事件在 NioEventLoop 中进行处理
serverBootstrap.group(new NioEventLoopGroup(workerCount, f));
```

java.nio.channels.SelectorProvider 在 Linux 下实现了基于 epoll 的事件通知工具，epoll 工具在 Linux 2.6 和更高版本的内核中可用。Netty 封装了对 SelectorProvider 的使用。

4.9 缓存

响应查询需要花费 CPU 的时间、内存，增加集群的处理能力有助于解决这个问题，但是过度配置的费用可能非常高。缓存经常是从优化工具箱中拉出的第一个工具。

缓存使用的内存总是有限的，因此需要使用一种算法来检测和替换没有价值的缓存。以下一些算法用于缓存项替换。

- Least Recently Used（LRU）：最近最少使用；
- Least Frequently Used（LFU）：最不经常使用；
- First In First Out（FIFO）：先进先出。

其中，最受欢迎的一个算法是最近最少使用（LRU）算法。研究表明，相比旧项目来说，新项目的使用率更高，LRU 就是基于这个原理，该算法保持跟踪项目的最后访问时间，清除有最早的访问时间戳的项目。

Elasticsearch 支持以下 3 种类型的缓存：节点查询缓存、分片请求缓存和字段数据缓存。

- 节点查询缓存是节点上所有分片共享的 LRU 缓存。它缓存在过滤器上下文中使用的查询结果，在以前的 Elasticsearch 版本中，由于这个原因也被称为过滤器缓存。过滤器上下文中的子句用于包含（或排除）结果集中的文档，但不影响评分。此外，许多过滤器计算速度非常快，特别是对于小的段，而其他过滤器则很少见。为了减少流失，节点缓存只包括以下过滤器：在最近 256 个查询中被多次使用属于超过 10000 个文档的段（或占文档总数 3%的过滤器，以较大者为准）。

- 分片请求缓存为每个分片独立地缓存查询结果，也使用 LRU 替换。默认情况下，请求缓存也限制子句，只缓存大小为 0 的请求（如聚合、计数和建议）。如果认为应该缓存查询，就可以在请求中添加 request_cache = true 标志。不是所有的子句都将被缓存，包含 now 的 DateTime 子句不会被缓存，在每次更新分片时都让分片请求缓存失效。这可能导致在频繁更新的索引中性能不佳。
- 字段数据缓存：当 Elasticsearch 计算字段上的聚合时，它会将所有字段值加载到内存中。因此，Elastisearch 中的计算聚合可以是查询中最“昂贵”的操作之一。字段数据缓存在计算聚合时保存字段值。虽然 Elastisearch 不跟踪命中/未命中率，但建议将其设置为足够大，以便将所有值都保存在内存中。

有许多集成可用于监控 Elasticsearch 缓存，如 Sematext 和 Datadog 都是一些比较常见的。但是如果只需要在开发过程中进行检查呢？

Elasticsearch 提供了许多方法来检查缓存利用率，_cat 节点 API 会在一次调用中给出上述所有的值。

```
# curl -XGET 'http://localhost:9200/_cat/nodes?v&h=id,queryCacheMemory,
queryCacheEvictions,requestCacheMemory,
requestCacheHitCount,requestCacheMissCount,flushTotal,flushTotalTime'
id   queryCacheMemory queryCacheEvictions requestCacheMemory
requestCacheHitCount requestCacheMissCount flushTotal flushTotalTime
yqrD         0b             0            0b                0
0                0          0             0s
```

上面的返回结果中，queryCacheMemory 表示使用的查询缓存内存数量，queryCacheEvictions 表示查询缓存替换次数；requestCacheMemory 表示使用的请求缓存数量；requestCacheHitCount 表示请求缓存命中次数；requestCacheMissCount 表示请求缓存缺失次数；flushTotal 表示刷新次数；flushTotalTime 表示刷新花费的时间。

4.10　分布式

Elasticsearch 使用主节点管理集群，主节点是集群中唯一可以更改集群状态的节点。这意味着如果主节点重新启动或关闭，那么将无法对群集进行任何更改。任何时刻集群中只能有一个主节点，但是为了避免单点失败，需要有多个候选主节点。

包括主节点在内的每个节点都知道每个文档所在的位置，并可将搜索请求直接转发到保存了相关数据的节点上。当搜索请求发送到一个节点上时，该节点就成为协调节点。这个节点的工作是将搜索请求广播给所有涉及的分片，并将它们的响应收集到全局排序的结果集中，以便返回给客户端。

4.11　Zen 发现机制

Elasticsearch 集群默认使用 Zen Discovery（Zen 发现机制）管理。

Zen 发现机制是 Elasticsearch 默认的内建模块，它提供了多播和单播两种发现方式，能够很容易地扩展至云环境。

Zen 发现机制是和其他模块集成的，例如所有节点间通信必须用 Transport 模块来完成。Transport 模块层是自己可以扩展的，thrift 也是一个 Transport 模块。

Elasticsearch 运行时会启动两个探测进程：一个进程用于从主节点向集群中其他节点发送 ping 请求来检测节点是否正常可用；另一个进程的工作相反，由其他的节点向主节点发送 ping 请求来验证主节点是否正常且忠于职守。

一个集群有一个唯一的名字，包含一个或者多个节点。集群会在所有的节点中自动选择一个作为主节点，如果主节点宕机了，则会自动选择另外一个节点作为主节点。经典的主节点选举算法是同行评审出版算法（peer-reviewed published algorithm）。

Elasticsearch 采用了一个简单的方法来选出主节点：即根据编号选择节点，较小的编号更有可能成为主节点。DiscoveryNode 类中记录了节点编号。选举算法的实现代码在 ElectMasterService.electMaster()方法中。

为了避免一个集群中存在不同的主节点，也就是避免“脑裂”，需要合理地设置 elasticsearch.yml 配置文件。

假设可以成为集群一部分的 Elasticsearch 节点的数量（是 Elasticsearch 的进程数量而不是物理机器的数量）是 N，那么在一个有 $N>2$ 个节点的集群上，可以设置 discovery.zen.minimum_master_nodes 的值不小于（$N/2$）+1。

理想的拓扑结构是有 3 个专用的主节点（即 master: true 并且 data:false），并且 discovery.zen. minimum_master_nodes 的设置为 2。这样无论集群中有多少数据节点，都不需要改变节点设置。

例如，主节点的配置如下：

```
node.master:true
node.data:false
discovery.zen.minimum_master_nodes:2
```

数据节点的配置如下：

```
node.master:false
node.data:true
discovery.zen.minimum_master_nodes:2
```

每个文档都保存在单独的主分片里。当对一个文档做索引的时候，首先对主分片做索引，然后在所有主分片的副本里做索引。默认一个索引有 5 个主分片，可以调整主分片的数量以控制一个索引中容纳文档的数量。索引创建之后，不可以更改主分片数，即使只在一台机器上安装 Elasticsearch，也可能会有 5 个独立的索引库。

每个主分片可以有 0 个或者多个副本，副本是主分片的复制品，有以下两个作用。

- 提高容错能力：如果主分片宕机，副本分片可以被提升至主分片。
- 提高性能：搜索访问可以分布在主分片和副本分片之间。

默认每个主分片有一个副本分片，但副本分片数量可以在已经存在的索引上动态调

整。在同一个节点上，副本分片不会被当做主分片启动。

我们用 3 个节点的集群举例说明索引分片的用处。假设在第一台计算机中存放索引分片 a、b、c，第二台计算机中存放索引分片 a、b、d，第三台计算机中存放索引分片 b、c、d。这样实现了提升索引整体容量的同时，也提升了性能和容错能力。

新增一个节点，Elasticsearch 会自动把索引数据同步到该新增的节点上。控制界面中显示的紫色的块表示正在迁移这部分数据。

主控节点管理 shard（分片）的分配，当有新的计算机进来或者有旧的计算机失效的时候，就会重新分配 shard。

依赖注入（Dependency Injection，DI）很好，因为一个节点有很多个索引，每个索引有很多个分片，每个分片是不同的 Guice 模块。Elasticsearch 使用 Google 开源的依赖注入框架 Guice（https://github.com/google/guice），而没有使用 Spring 实现依赖注入的原因是：Spring 需要配置文件，用起来太笨重。Elasticsearch 直接把 Guice 的源码放入了自己的 org.elasticsearch.common.inject 包内。

4.12　联合搜索

Elasticsearch 中可以使用跨群集搜索来执行联合搜索。Elasticsearch 5.3.0 版本以后，可以通过 search.remote 命名空间下的集群更新设置 API 注册远程集群。每个群集都由群集别名和用于发现属于远程群集的其他节点的种子节点列表进行标识。代码如下：

```
PUT _cluster/settings
  {
   "persistent": {
     "search": {
       "remote": {
         "cluster_one": {
           "seeds": ["remote_node_one:9300"]                    //集群一的种子节点
         },
         "cluster_two": {
           "seeds": ["remote_node_two:9300"]                    //集群二的种子节点
         }
       }
     }
   }
  }
```

一旦注册了一个或多个远程集群，就可以使用_search API 对其索引执行搜索请求。与本地索引相反，远程索引必须增加群组别名前缀来消除歧义，如 cluster_two:index_test。

每当搜索请求扩展到远程集群上的索引时，协调节点通过每个集群发送一个_search_shards 请求来解析远程集群上这些索引的分片。一旦获取了分片和远程数据节点，就可以像执行本地集群上的搜索一样，使用完全相同的代码路径，显著提高了可测试性和鲁棒性。

我们可以通过 search.remote.connect 设置来控制哪些节点可以作为跨集群搜索请求的协调节点。这对于控制集群中的哪些节点可以向远程集群发送请求是有用的。如果不允许连接到远程集群的节点接收涉及远程集群的搜索请求，则会返回错误。

4.13 JVM 字节码

Java 程序运行在专门为它定义的虚拟机上，该虚拟机叫做 Java Virtual Machine（简称 JVM）。JVM 本身并没有定义 Java 程序设计语言，它只定义了包含 JVM 指令集的 class 文件的格式及一个符号表等。

根据 CPU 的寻址空间，可以将 CPU 分为 32 位和 64 位的。JVM 为这两种 CPU 做了专门的版本，字长是 32 位和 64 位的 JVM。为了实现这样跨平台运行的想法，每种常用的 CPU 和操作系统组合都有现成的 JVM 实现，如图 4-10 所示。

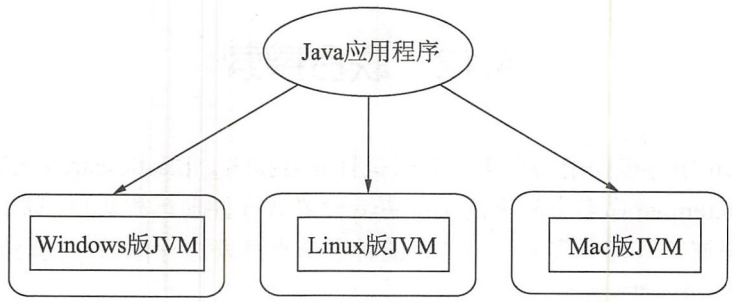


图 4-10　运行在 JVM 上的 Java 应用程序

可以在 Windows 平台编译一个 Java 程序，并且在 Linux 平台运行它，因为 Windows 和 Linux 都支持 JVM 实现。Linux 平台下的指令集可能和 Java 编译器生成的字节码不一样。一个 JVM 可以一次性地解释字节码，或者把字节码编译成它所运行的本地代码，这种优化技术叫做 JIT（Just-In-Time）。Java 字节码程序可以运行在任何有字节码解释器的程序上，如图 4-11 所示。

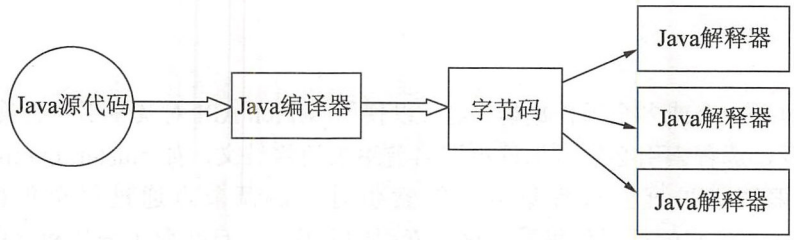


图 4-11　一次编译随处运行的 Java 应用程序

JVM 定义了一套通用的虚拟指令集，即 JVM 指令集。JVM 是基于栈结构的计算机。JVM 指令集中的相关操作数在栈顶运算。例如，如果解释器要执行整数加法，就是从栈顶弹出两个整数进行加法运算，最后将结果压入栈顶。

4.13.1　编译代码

Java 源代码首先通过 JavaCompiler 编译成字节码存放在 Class 文件中，在运行前 JVM 还可以把 Class 文件编译成机器码。

使用 JavaCompiler 编译源代码：

```
//先检查 java.home/lib 下的 tools.jar 文件，该文件必须存在
System.out.println(System.getProperty("java.home"));
JavaCompiler compiler=ToolProvider.getSystemJavaCompiler();

System.out.println("JavaCompiler: " + compiler);
int results = compiler.run(null, null, null, "f:/test/Cat.java");
System.out.println((results == 0) ? "编译成功" : "编译失败");
```

在 Java 9 中，编译出的代码不再是使用内置的即时（JIT）编译器专门创建的。Java 9 为使用 Java 编写的 JIT 编译器指定了一个新的接口（JVMCI）。这意味着任何人都可以发布一个可以轻松连接到虚拟机的 JIT 编译器。编译器可以是系统中独有的 JIT 编译器，也可以与分层模式下的内置 JIT 编译器一起使用。在第一种情况下，编译器（就像任何应用程序一样，只是 Java 代码）解释并最终编译自己；然而，在后一种情况下，编译后的代码可以用于任何编译层。与其他虚拟机一样，Hotspot 已经默认为分层编译模式。在这种模式下，一个方法是通过解释开始的，但是在达到特定数量的执行后，由客户端编译器编译（快速生成代码，但只是相当简单的优化）。最后一旦发生了足够数量的执行，代码由服务器端编译器编译（速度相对较慢，但积极地优化）以实现最佳性能。自定义 JIT 可以在任何级别插入，这意味着它既可以用作中间层，也可以用作最后一层。

Oracle JDK 9 中的 JAOTC（Java Ahead-of-time Compiler）可以把指定的 Java Class 文件/module 文件提前编译成机器码。在 Linux/x86-64 中，生成的机器码被包装在 ELF 格式的文件中，用 Linux 自带的 objdump 就可以查看它的内容。

尽管虚拟机会立即开始使用提前编译的代码（绕过解释），但是代码仍然会被最终编译以实现更好的性能。这个功能主要是为了减少启动时间，提高执行过少的方法的性能，因为它们不能进行 JIT 编译。

4.13.2　同步相关指令

JVM 中只有两个指令和同步相关，一个是进入 monitor，还有一个是退出 monitor。monitorenter 和 monitorexit 用于同步语句块。monitorenter 进入到一个同步语句块，该语句块被锁住，monitorexit 离开这个语句块，解锁。

每个对象都有一个对应的 monitor，执行 monitorenter 的线程获得对象引用的所有权。如果有其他线程获取了这个对象的 monitor，当前的线程就要一直等待，直到这个对象解锁，然后再试着得到所有权。一个线程不会被自己阻塞，如果当前线程已经拥有一个对象上的锁，则执行 monitorenter 会让计数器递增，当计数器返回 0 时，锁才会被释放。

一个 monitorenter 指令可以和一个或多个 monitorexit 指令一起使用，实现一个 synchronized 语句。但是 monitorenter 和 monitorexit 指令没有用于执行同步方法，虽然可以用它们来提供相同的锁语义。一个同步方法调用的监控项目，通过 Java 虚拟机的方法调用指令进行隐式地处理。可以在同步方法的定义上设置 ACC_SYNCHRONIZED 标签，所以方法的实际字节码看不出来。

类似的效果也发生在 volatile 属性上：它只是简单地设置属性的 ACC_VOLATILE 标志，访问这个属性的代码使用相同的字节码，只是行为稍有不同。

同步方法和同步代码块基本上是等价的。同步方法的写法如下：

```
public synchronized void method() {     // 从这里阻塞"this"....
    ...
    ...
    ...
} // 到这里
```

同步代码块的写法如下：

```
public void method() {
    synchronized( this ) {                  //从这里阻塞"this" ....
        ...
        ...
        ...
    }                                       //到这里
}
```

4.14 本章小结

关于 Lucene 的 Java Doc 说明文档可参考网址 http://lucene.apache.org/core/documentation.html。

Doug Cutting 在 1999 年开发 Lucene 以前，使用 C++开发过搜索引擎，Lucene 是他写的第一个 Java 软件。在后来的十多年里，Lucene 越来越流行，成为开源组织 Apache 基金会的项目，并在维基百科网站等项目中得到了广泛应用。Doug Cutting 后来开发的 MapReduce 的 Java 版本 Hadoop 也同样成功，也因此进入了 Apache 基金董事会，并在 2010 年成为董事会主席。

可以使用 Luke（网址为 https://github.com/DmitryKey/luke）分析本地硬盘上的 Elasticsearch 索引，使用 luke.bat 或 luke.sh 运行 Luke，然后就可以在/indexname/0/index/这样的路径中打开索引。

某些物种进化到一定程度以后，就开始逐步收敛。Lucene 6 以后，使用它的代码不再有什么变化了。

Ant 可以自动化打包逻辑，Maven 也可以自动化打包。相比于 Ant，Maven 多做的事是帮你下载 jar 包，而 Gradle 既能自动下 jar 包，又能自己写脚本。

Elasticsearch 早期的版本使用 JGroups 实现多播。JGroups 是一个用于方便集群开发的组件，它依赖组播。

Netty 是一个高性能、事件驱动的异步非堵塞的 I/O 框架。Elasticsearch 使用 Netty 作为 HTTP 容器，可以灵活地通过插件为 Netty 管道添加安全性。

直到 Elasticsearch 5 版本为止，仍然采用 HTTP/1.1，没有采用性能更好的 HTTP/2。

第 5 章　提高搜索相关性

搜索引擎需要把符合查询条件的文档按相关度排序后输出。评估查询词和文档相关性的方法叫做检索模型。当前有 BM25 和学习评分两种流行的方法。

对于一个封闭的文档集合，可以使用准确率和召回率等指标来评价检索模型。其中：

- 准确率=返回结果中相关文档数目/返回结果的数目；
- 召回率=返回结果中相关文档数目/所有相关文档数目。

下面从最基本的向量空间检索模型开始介绍。

5.1　向量空间检索模型

向量空间模型（Vector Space Model，VSM），是一个把文档表示成索引词形成的向量的代数模型，如图 5-1 所示。

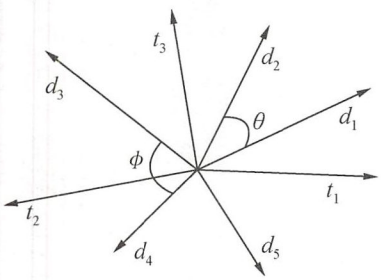


图 5-1　向量空间

如果两个向量夹角为 0°，则相似度为 100%；如果两个向量夹角为 180°，则相似度为 0。

文档 d 对于查询词 q 的 VSM 分值是带权重的查询向量 $V(q)$和文档向量 $V(d)$的夹角余弦（Cos）相似度：

$$\text{cosine-similarity}(q,d)=\frac{V(q)\times V(d)}{\|V(q)\|\times\|V(d)\|}=\frac{V(q)}{\|V(q)\|}\times\frac{V(d)}{\|V(d)\|}$$

式中：

- 查询向量 $V(q)=<w(t_1, q), w(t_2, q),\cdots, w(t_n, q)>$；

- 文档向量 $V(d) = <w(t_1, d), w(t_2, d), \cdots, w(t_n, d)>$；
- $V(q) \cdot V(d)$是两个带权重的向量的点积，计算方法是 $V(q) \cdot V(d) = w(t_1, q) \cdot w(t_1, d) + w(t_2, q) \cdot w(t_2, d) + \cdots + w(t_n, q) \cdot w(t_n, d)$；
- $\|V(q)\|$和$\|V(d)\|$表示欧几里得范数。例如，$\|V(d)\| = \sqrt{t_1^2 + t_2^2 + \cdots + t_n^2}$，这里的 t 是文档 d 中出现的词的权重。

式中分母涉及向量的长度，用来归一化。向量长度归一化的方法是：每个分量除以它的长度，这里使用 L2 范数计算向量长度：

$$\|\vec{x}\|_2 = \sqrt{\sum_i x_i^2}$$

这里使用 L2 范数把文档归一化成为单位向量，Cos(q,d)是 q 的单位向量和 d 的单位向量点乘积。如图 5-2 是一个简单的图示例子。

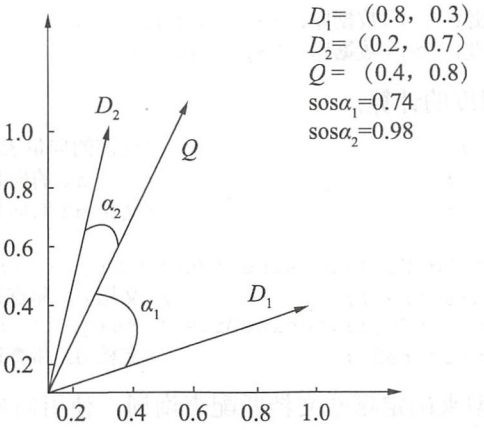


图 5-2　查询相似度

计算向量长度的实现代码如下：

```
public static double vectorLength(int[] vector) {
    double sumOfSquares = 0d;
    for (int i = 0; i < vector.length; i++) {
        sumOfSquares = sumOfSquares + (vector[i] * vector[i]) ;
                                                    //平方求和
    }

    return Math.sqrt(sumOfSquares);                 //开根号
}
```

计算两个向量之间点积的代码如下：

```
public static int scalarProduct(int[] one, int[] two){
    int result = 0;
    for (int i = 0; i < one.length; i++) {      //两个输入向量长度是相同的
        result += one[i] * two[i];
    }
```

```
        return result;
    }
```

根据点积和向量长度计算夹角余弦的代码如下：

```
public static double cosineOfVectors(int[] one, int[] two){
    double denominater = (vectorLength(one) * vectorLength(two));
                                                    //分母
    if (denominater == 0) {
        return 0;
    } else {
        //两个向量之间的点积除以两个向量的长度乘积
        return (scalarProduct(one, two) / denominater);
    }
}
```

举例应用计算代码：

```
查询 q：(<开源：1>,<数据库：2>)
      文档 d1：(<开源：1>,<数据库：3>,<PostgreSQL：1>)
      文档 d2：(<商业：1>,<数据库：2>,<Oracle：1>)
```

查询和文档进行向量相似度的计算：

```
int[] q = { 1, 2 };                          //查询的向量表示
int[] d1 = { 1, 3 };                         //文档 d1 的向量表示
int[] d2 = { 0, 2 };                         //文档 d2 的向量表示

double score1 = VectorUtils.cosineOfVectors(q, d1);
System.out.println(score1);                  //文档 d1 和查询 q 的夹角余弦值 0.99
double score2 = VectorUtils.cosineOfVectors(q, d2);
System.out.println(score2);                  //文档 d2 和查询 q 的夹角余弦值 0.89
```

Lucene 使用布尔模型来确定哪些文档匹配查询词，使用向量空间模型对这些文档评分。评分算法中的向量空间模型使用 Tf-idf 计算权重。

TF（Term Frequence）代表词频，IDF（Invert Document Frequence）代表文档频率的倒数。比如“的”在 100 篇索引文档中的 40 篇文档中出现过，则文档频率 *DF*（Document Frequence）是 40，*IDF* 是 1/40。“的”在第一篇文档中出现了 15 次，则 *TF* • *IDF*(的) = 15 ×1/40=0.375。另外一个词“户口”在 100 篇文档中的 5 篇文档中出现过，则 *DF* 是 5，*IDF* 是 1/5。“户口”在第一篇文档中出现了 5 次，则 *TF* • *IDF*(户口) = 5×1/5=1。结果是：*TF* • *IDF*(户口)> *TF* • *IDF*(的)。对给定的词 *t* 和文档（或者查询）*x*，*Tf*(*t*,*x*) 的值和词 *t* 在 *x* 中出现的次数正相关，而 *idf*(*t*)的值和索引文档集合中包含词 *t* 的次数负相关。

输入向量由整数改成浮点数计算夹角余弦，代码如下：

```
public double cosineSimilarity(double[] docVector1, double[] docVector2) {
    double dotProduct = 0.0;                 //存两个向量的点积值
    double magnitude1 = 0.0;                 //第一个向量的范数
    double magnitude2 = 0.0;                 //第二个向量的范数
    double cosineSimilarity = 0.0;

    for (int i = 0; i < docVector1.length; i++) {
```

```
            dotProduct += docVector1[i] * docVector2[i];    //计算 a.b
            magnitude1 += Math.pow(docVector1[i], 2);       //计算 (a^2)
            magnitude2 += Math.pow(docVector2[i], 2);       //计算 (b^2)
        }

        magnitude1 = Math.sqrt(magnitude1);                 //计算 sqrt(a^2)
        magnitude2 = Math.sqrt(magnitude2);                 //计算 sqrt(b^2)

        if (magnitude1 != 0.0 | magnitude2 != 0.0) {
            //得到夹角余弦值
            cosineSimilarity = dotProduct / (magnitude1 * magnitude2);
        } else {
            return 0.0;
        }
        return cosineSimilarity;
    }
```

5.2　BM25 检索模型

可以认为打上了和查询词同样标签的文档是相关文档。但很多时候，猜测文档是否有相关内容是没有把握的。所以可以用概率来量化这种不确定性，可以把信息检索作为分类问题，一类是相关文档 R，还有一类是无关的文档 NR。根据贝叶斯判别规则，如果 $P(R|D)>P(NR|D)$，则 D 是相关的文档；如果 $P(R|D)<P(NR|D)$，则 D 是不相关的文档。例如，$P(R|D)=0.8$，$P(NR|D)=0.2$，则 D 是和用户查询相关的文档。如图 5-3 所示，把“新生婴儿入户须知”这个索引库中的文档分成了相关文档和不相关文档。

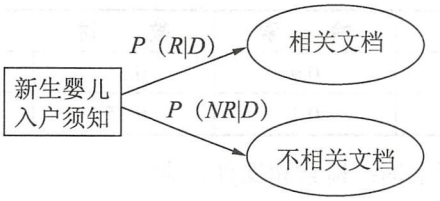


图 5-3　把信息检索看成分类问题

如果知道相关文档集合，就能够计算出 $P(D|R)$。例如，如果知道某个词在相关文档集合中频繁出现的次数，然后给定一个新文档，就能直接计算出该文档中词的组合有多大可能性出现在相关文档集合中。

使用贝叶斯公式估计概率：

$$P(R \mid D)=\frac{P(D \mid R)P(R)}{P(D)}$$

比较 $P(R|D)$和 $P(NR|D)$的值。如果满足 $P(R|D)\ P(R)>P(D|NR)\ P(NR)$，则把文档分到相关类中。把一个文档分类成相关的条件，可以写成：

$$\frac{P(D|R)}{P(D|NR)} > \frac{P(NR)}{P(R)}$$

左边的式子叫做似然比，需要计算 $P(D|R)$和 $P(D|NR)$。为了简化计算，我们把文档表示成词的组合，用词概率估计 $P(D|R)$和 $P(D|NR)$。

可以用一个二值特征的向量表示文档的特征，表示文档中出现或者不出现某个词。把文档表示成二项特征组成的向量，$D=(d_1, d_2, \ldots, d_t)$，如果词 i 出现在文档中，则 $d_i=1$，否则就是 0。如果假设词都是独立出现的，则 $P(D|R)$可以用词概率的乘积 $\prod_{i=1}^{t} P(d_i \mid R)$ 计算。因为这个模型假设词独立出现，而且使用文档的二项特征，所以叫做二项独立模型。

假设索引库包含 5 个词，某文档 D 根据二元假设，表示为{1,0,1,0}，其含义是这个文档出现了第 1 个、第 3 个和第 5 个词，但不包含第 2 个和第 4 个词。

我们用 P_i 来代表第 i 个词在相关文档集合内出现的概率，于是在已知相关文档集合的情况下，文档 D 相关的概率为：

$$P(R|D)=P_1\times(1-P_2)\times P_3\times(1-P_4)\times P_5$$

其中的 $1-P_2$ 代表了第 2 个词不出现在相关文档中的概率，因为 $P(t_2|R)+P(\overline{t}_2|R)=1$。

为了计算 $P(D|NR)$，假设用 S_i 代表第 i 个词语或单词在不相关文档集合内出现的概率，于是在已知不相关文档集合的情况下，观察到文档 D 的概率为：

$$P(D|NR)=S_1\times(1-S_2)\times S_3\times(1-S_4)\times S_5$$

例如，查询“信息 检索 教程”的所有词项在相关、不相关情况下的概率说明，p_i、s_i 分别如表 5-1 所示。

表 5-1　概率说明表

词　项	信　息	检　索	教　材	教　程	课　件
R中的概率p_i	0.8	0.9	0.3	0.32	0.15
NR中的概率s_i	0.3	0.1	0.35	0.33	0.10

假设文档 D_1 中只有两个词：检索和课件，则

$$P(D|R)=(1-0.8)\times 0.9\times(1-0.3)\times(1-0.32)\times 0.15$$

$$P(D|NR)=(1-0.3)\times 0.1\times(1-0.35)\times(1-0.33)\times 0.10$$

$$P(D|R)/P(D|NR)=4.216$$

返回到似然比。使用 P_i 和 S_i 得到分值：

$$\frac{P(D|R)}{P(D|NR)}=\prod_{i:d_i=1}\frac{p_i}{s_i}\prod_{i:d_i=0}\frac{1-p_i}{1-s_i}$$

其中，$\prod_{i:d_i=1}$ 的意思是在文档向量中值为 1 的词对应的乘积。把上面的公式转换下：

$$\prod_{i:d_i=1}\frac{p_i}{s_i}\prod_{i:d_i=0}\frac{1-p_i}{1-s_i}=\prod_{i:d_i=1}\frac{p_i}{s_i}\left(\prod_{i:d_i=1}\frac{1-s_i}{1-p_i}\prod_{i:d_i=1}\frac{1-p_i}{1-s_i}\right)\prod_{i:d_i=0}\frac{1-p_i}{1-s_i}$$

$$=\prod_{i:d_i=1}\frac{p_i(1-s_i)}{s_i(1-p_i)}\prod_i\frac{1-p_i}{1-s_i}$$

第二项在所有定义向量维度的词上运算，因此对任何文档来说，值都是一样的，对文档评分时可以忽略这一项。

因为多个很小的数相乘可能会导致精度丢失或者向下溢出成为 0，所以对计算公式取 log，这样评分公式变成了

$$\sum_{i:d_i=1}\log\frac{p_i(1-s_i)}{s_i(1-p_i)}$$

如果存在相关性反馈，则可以得到相关文档和无关文档集合。也就是说，给定用户查询，如果可以确定哪些文档构成了相关文档集合，哪些文档构成了不相关的文档集合，那么就可以利用表 5-2 所列出的数据来估算单词概率。

表 5-2　某个查询的词出现情况的相依表

	相关文档	不相关文档	文档总数
d_i=1	r_i	n_i-r_i	n_i
d_i =0	R-r_i	N- n_i -R + r_i	N-n_i
文档总数	R	N-R	N

表 5-2 中第 3 行的 N 为文档集合总共包含的文档个数，R 为相关文档的个数，于是 N-R 就是不相关文档集合的大小。对于某个词语或单词 d_i 来说，假设包含这个词语的文档数量共有 n_i 个，而其中相关文档有 r_i 个，那么不相关文档中包含这个单词的文档数量则为 n_i - r_i。再考虑表中第 2 列，因为相关文档个数是 R，而其中出现过单词 d_i 的有 r_i 个，那么相关文档中没有出现过这个单词的文档个数为 R-r_i 个，同理，不相关文档中没有出现过这个单词的文档个数为(N-R)-(n_i-r_i)个。从表 5-2 中可以看出，如果假设我们已经知道 N、R、n_i、r_i 的话，那么其他参数可以靠这 4 个值推导出来。

采用最大似然估计，计算 $p_i=\frac{r_i}{R}$，$s_i=\frac{n_i-r_i}{N-R}$。为了避免 r_i 是 0 导致的 log0 无法计算的问题，可以采用相关文档和不相关文档都加 0.5 的平滑方法。这样得到 $p_i=\frac{r_i+0.5}{R+1}$，$s_i=\frac{n_i-r_i+0.5}{N-R+1}$。把这些值放入打分公式中得到

$$\sum_{i:d_i=q_i=1}\log\frac{(r_i+0.5)/(R-r_i+0.5)}{(n_i-r_i+0.5)/(N-n-R+r_i+0.5)}$$

这个打分公式没有考虑词频，相关度比考虑词频的公式低 50%。

Okapi BM25（简称 BM25）是一种相关性排序函数，适用于搜索引擎根据与给定搜索查询的相关性对匹配文档进行排序。

BM25 是一个基于单词集合的检索函数，它依据出现在每个文档中的查询词对匹配文档集合排序，而不管查询词在文档内相互之间的联系。它不是一个单一的函数，实际上是有略微不同的组件和参数变化的一群函数的集合。一个最典型的具体函数如下：

假定有一个查询词组 Q，含有关键词 $q_1,\cdots,q_n$，用 BM25 给文档 D 评分的公式是

$$SCORE(D,Q)=\sum_{i=1}^{n} IDF(q_i)\times\frac{f(q_i,D)\cdot(k_1+1)}{f(q_i,D)+k_1\cdot(1-b+b\cdot\frac{|D|}{avgd1})}$$

其中，$f(q_i,D)$是检索词 q_i 在文档 D 中的频率，$|D|$是文档 D 以单词为单位的长度，$avgd1$ 是从抽取出的文档的文本集合的平均文档长度。k_1 和 b 是自由参数，通常选择 k_1=2.0 和 b=0.75。$IDF(q_i)$是检索词 q_i 的 IDF（文档频率倒数）权重。$IDF(q_i)$的一般计算公式是

$$IDF(q_i)=\log\frac{N-n(q_i)+0.5}{n(q_i)+0.5}$$

这里，N 是集合中文档的总数，$n(q_i)$是包含 q_i 的文档个数。

5.2.1 使用 BM25 检索模型

基本上可以在索引设置中定义类似于自定义分析器的自定义 BM25 相似度，也就是定义 b 和 k_1 的值。例如：

```
# curl -XPUT "http://<server>/<index>" -d '
{
  "settings": {
    "similarity": {
      "custom_bm25": {
        "type": "BM25",          //相似性类型为 BM25
        "b":    0 ,              //设定 b 的值
         "k1" : 0.9              //设定 k1 的值
      }
    }
  }'
```

5.2.2 参数调优

为了在 BM25 中调整参数 b 和 k_1（这些参数对数据集非常依赖），简单的方法是：调整参数，然后检查结果，如果不满意，更改参数并再次测试结果。也可以使用像遗传算法或蚁群算法（Ant colony optimization）这样的启发式算法自动调整参数。

TF 归一化调优的经典方法是旋转归一化方法，但这个方法存在集合依赖问题，可以

下载 TREC 提供的数据集进行自动调参。数据集中的 Ad hoc Search 针对一个固定的文档集合，根据用户输入的查询问题会返回相关性降序输出的文档列表。

Jenetics（下载地址是 https://github.com/jenetics/jenetics）是一个以 Java 语言编写的遗传算法库。它被设计成明确地分离算法的几个概念，如，基因、染色体、基因型、表型、群体和适应度函数（Fitness Function）。

5.3　学习评分

可以利用用户的点击日志分析出哪些文档和查询词最相关，根据用户搜索行为调整搜索结果排序，此外，还可以通过社交网络判断文档相关度。

学习评分（Learning to Rank）采用机器学习方法训练出的采用多种特征的模型用来对文档相关度进行评分。它是一种有监督或半监督的机器学习问题，其目标是从训练数据自动构建评分模型，这样恶意用户更难以操作排名。

我们可以使用流行度数据用于训练。例如，访问量、用户在页面停留了多久等。

5.3.1　基本原理

可以人工标注出一个理想的文档相关性排序，然后采用一种学习评分算法学习出模型。LambdaMART 是一种当前流行的学习评分算法，是从 Pairwise 方法中逐渐发展起来的方法。Pairwise 方法的主要思想是将排序问题形式化为二元分类问题；Pairwise 方法通过考虑两两文档之间的相对相关度进行排序。例如，文档 X 比文档 Y 更相关，还是更不相关，这是一个二元分类问题。其目标是最小化反转的排名，也就是让损失最小。RankNet 算法其实是一个 Pairwise 方法，它使用交叉熵作为损失函数，损失函数的值越低说明机器学得的当前排序越趋近于理想排序。RankNet 算法可以使用神经网络模型，也可以使用渐进梯度回归树（Gradient Boost Regression Tree，简称 GDBT）模型。

如果 GDBT 模型求解过程使用求梯度的 Lambda 方法，就是 LambdaMART 算法。这里的 MART（Multiple Additive Regression Tree），也就是 GBDT（GradientBoosting Decision Tree）。Lambda 的含义是一个待排序的文档下一次迭代应该排序的方向（向上或者向下）和强度。

在实施学习排名时，需要注意以下几点：

- 通过分析测量用户认为相关的内容，为查询建立一个评估列表，将文档分级为完全相关、中等相关、不相关。
- 假设哪些特征可能有助于预测相关性，如特定字段匹配的 $TF \cdot IDF$ 值、新鲜度、搜索用户的个性化等。
- 训练一个模型，可以准确地映射特征到相关性的分数。

- 将模型部署到你的搜索基础架构上，使用它对生产系统中的搜索结果进行排名。

机器学习评分包 RankLib（网址为 https://sourceforge.net/p/lemur/wiki/RankLib/）用于 Lemur 搜索引擎，但也可以修改后和 Lucene 一起使用。

可以直接在命令行运行 RankLib.jar：

```
> java -jar RankLib.jar
```

5.3.2 准备数据

本节以搜索电影作为例子。这里使用电影数据（https://github.com/holgerbrandl/themoviedbapi），先在电影数据库网站（https://www.themoviedb.org）申请 API，然后从 https://jcenter.bintray.com/网站下载依赖的 jar 包。

如果使用 Maven，那么可以手动将此 repo 添加到 pom.xml 或 settings.xml 中：

```
<repositories>
    <repository>
        <id>jcenter</id>
        <releases>
            <enabled>true</enabled>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
        <url>https://jcenter.bintray.com/</url>
    </repository>
</repositories>
```

搜索电影数据库的代码如下：

```
TmdbApi tmdb = new TmdbApi(apiKey);                    //使用 Key 创建 API
TmdbSearch search = tmdb.getSearch();                  //得到查询对象
String keyWords = "Rambo";                             //查询词
//返回搜索结果页
MovieResultsPage movieResultsPage =
    search.searchMovie(keyWords, null, null, false, null);
//得到电影列表
List<MovieDb> movies = movieResultsPage.getResults();
System.out.println(movies.size());                     //输出返回的电影数量
System.out.println(movies);                            //按相关度输出电影列表
```

首先创建一个判断列表。一般用一个 0～4 的数值表示，0 表示不相关，4 表示完全相关。

然后考虑搜索 rambo。如果 doc_1234 是电影 Rambo，而 doc_5678 是 Turner and Hooch，那么可以做出以下两个判断：

```
4,rambo,doc_1234 # 电影 "Rambo" 是查询词"rambo"的精确匹配（第 4 等级）
0,rambo,doc_5678 # 电影"Turner and Hooch"则与查询词"rambo"完全不相关（0 级）
```


为了使用 Ranklib 预测相关性等级，需要做一些预处理来检查查询词和文档，并生成一组定量特征。这里的特征可以是测量查询、文档，或测量查询和文档之间的关系数值。

例如，电影的特征可能包括：

- 搜索关键字是否与标题字段匹配（我们称之为 titleScore）；
- 搜索关键字是否与描述字段匹配（descScore）；
- 电影的受欢迎程度（受欢迎程度）；
- 电影评级（评级）；
- 在搜索过程中使用了多少个关键字（numKeywords）。

可以任意决定特征的编号。例如，特征 1 是查询关键字在电影标题中出现的次数，而特征 2 可能对应于电影概览字段中关键字出现的次数。

这种训练集的通用文件格式如下：

```
等级 qid:<queryId> 1:<feature1Val> 2:<feature2Val>... #评论
```

举个例子，当查看查询 rambo 时，将其称为查询 Id 1，注意以下特征值。

- 特征 1：Rambo 在电影 Rambo 的标题中出现 1 次；0 次在 Turner and Hootch。
- 特征 2：Rambo 在电影 Rambo 的描述字段出现 6 次；0 次在 Turner and Hootch。

然后上面的判断表就变成了：

```
4    qid:1    1:1    2:6
0    qid:1    1:0    2:0
```

qid:1 的相关电影（Rambo）具有比不相关电影（Turner and Hootch）的匹配更高的特征 1 和 2 的值。

一个完整的训练集用在成千上万或更多查询上的分级文档来表示这个想法：

```
4    qid:1    1:1    2:6
0    qid:1    1:0    2:0
4    qid:2    1:1    2:6
3    qid:2    1:1    2:6
0    qid:2    1:0    2:0
...
```

训练的目的是生成一个函数（这里也松散地称之为模型），它接受输入特征 1…n 并输出相关性等级。下载 Ranklib 以后，可以训练出一个模型文件如下：

```
> java -jar bin/RankLib.jar -train train.txt -ranker 6 -save mymodel.txt
```

上面的命令训练数据 train.txt 已生成 LambdaMART 模型（Ranker 6），将模型的文本表示输出到 mymodel.txt 中。一旦有了一个好的模型，就可以用它作为排名函数产生相关性分数了。

5.3.3 Elasticsearch 学习排名

elasticsearch-learning-to-rank（下载地址是 https://github.com/o19s/elasticsearch-learning-to-rank）以插件方式提供了让 Elasticsearch 支持学习排名的方法。例如，要在 Elasticsearch 5.4.0 上安装插件的 0.1.2 版，请执行以下操作：

```
./bin/elasticsearch-plugin install http://es-learn-to-rank.labs.o19s.
com/ltr-1.0.0-RC2-es5.6.4.zip
```

让 ltr 查询使用该模型来打分。下面的 dummy 是打分过的模型，每个“特征”都会反映出在训练时使用的查询。

```
GET /foo/_search
{
    "query": {
        "ltr": {                                    //学习排名类型的查询
         "model": {
             "stored": "dummy"                      //指定模型名称为 dummy
         },
         "features": [{
             "match": {
                 "title": userSearchString          //指定查询词
             }
         }]
        }
    }
}
```

5.4 查询意图识别

用户提交给搜索引擎的查询都是有意图的。用户搜索意图一般有如下几个常见的类别。

- 导航类：用户不知道要访问的网站的网址，所以用搜索引擎查找。例如，输入 Elasticsearch 访问 Elasticsearch 官方网站。
- 信息类：想知道某个问题的答案，如“如何才能减肥”。
- 资源类：这种类型的搜索目的是希望能够从网上获取某种资源。例如，输入“谷歌浏览器”下载这个软件，或者输入一本书的书名找到购买网址。

可以使用查询意图模板识别查询意图。例如：

```
public class Intent {                               //用户查询意图
public String type;                                 //意图类型
public PairListString args;                         //参数
}
```

可以通过查询串估计查询意图，并产生相应的查询对象。生成查询实例的代码如下：

```
String key = "keyword";                             //键
```


```
String val = "Elasticsearch";                    //值

PairListString intentArgs = new PairListString(1);
intentArgs.addPair(key, val);
Intent queryIntent = new Intent("navigate", intentArgs);
                                                 //创建一个导航类查询实例
```

可以使用问句模板匹配用户查询词。例如，问句模板“<菜名>的做法”匹配用户查询“青椒炒牛肉的做法”，匹配代码如下：

```
String query = "青椒炒牛肉的做法";                  //用户查询串

QueryTemplate template = new QueryTemplate();    //问句模板引擎
template.addWord("青椒炒牛肉", "菜名");            //加入菜名

String right = "<菜名>的做法";                    //加入问句模板
template.add(right);

Intent queryIntent = template.match(query);      //返回匹配出的查询意图
```

英文模板 recipe for $X($X 的菜谱)匹配用户查询 recipe for beef stirfry(炒牛肉的菜谱)，这里的$X 表示一个变量。

5.5　图像特征提升检索体验

越来越多的网页开始有配图介绍。搜索引擎系统可以提取网页中的图像特征来提升用户的搜索体验。例如，当查询人名的时候，包含人脸图像的网页结果会有更高的权重。

可以利用 OpenCV 中的级联分类器模型检测图像中的人脸。为了在 Java 中使用 OpenCV，需要先加载操作系统对应的动态链接库。

首先介绍在 Windows 开发环境下使用 OpenCV 3.x。从 OpenCV 官方网站（http://opencv.org）下载 OpenCV 库（3.x 版）opencv-3.x.0-vc14.exe，然后在选择的位置提取下载的 OpenCV 文件。为了用 Java 调用 OpenCV，只需要两个文件，分别是位于\opencv\build\java 的 opencv-3xx.jar 文件和位于\opencv\build\java\x64（用于 64 位系统）或 \opencv\build\java\x86（对于 32 位系统）的 opencv_java3xx.dll 库。每个文件的 3xx 后缀是当前 OpenCV 版本的快捷方式，例如，OpenCV 3.0 是 300，而 OpenCV 3.2 是 320。在 Eclipse 中把这两个文件加到用户库中，其中的 dll 加到本地库路径下。

首先记得加载 dll，然后才能使用 OpenCV：

```
System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
System.out.println("OpenCV Version:" + Core.VERSION);
                                                 //输出版本号以验证是否加载成功
```

OpenCV 使用 Mat 对象表示图像，Mat 是一个多维的密集数据数组。因为 OpenCV 不支持读入所有的图片格式，所以首先需要验证是否正确读入：

```
String inputImg = "info1.jpg";
Mat image = Imgcodecs.imread(inputImg);
System.out.println("empty?"+image.empty());     //如果 image 是空的，则读入错误
```

使用 OpenCV 中的级联分类器来实现人脸检测的代码：

```
//加载 dll
System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
CascadeClassifier faceDetector = new CascadeClassifier(
        "haarcascade_frontalface_alt.xml");     //创建识别人脸的级联分类器
//待识别的图片读入到矩阵对象
Mat image = Imgcodecs.imread("info1.jpg");

MatOfRect faceDetections = new MatOfRect();     //识别结果放入矩阵数组
faceDetector.detectMultiScale(image, faceDetections);

System.out.println(String.format("Detected %s faces",
        faceDetections.toArray().length));      //输出检测出来的人脸数量

//把识别出的人脸用矩形框出来
for (Rect rect : faceDetections.toArray()) {
Scalar color = new Scalar(0, 255, 0);           //用颜色对象设定矩形框的颜色
Imgproc.rectangle(image, new Point(rect.x, rect.y), new Point(
        rect.x + rect.width, rect.y + rect.height), color, 3);
}

//写入新的图像文件
String filename = "info1-FaceDetector.jpg";
System.out.println(String.format("Writing %s", filename));
Imgcodecs.imwrite(filename, image);
```

这里使用自带的模型文件，也可以使用 opencv_traincascade 自己训练出的 XML 模型文件。

为了支持在多个操作系统下跨平台运行，可以先判断操作系统类型，然后加载对应的动态链接库。为了得到 Linux 下的动态链接库，可以从源代码编译生成 libopencv_java320.so，或者下载编译好的 libopencv_java320.so 文件。

判断操作系统类型的代码如下：

```
public final class OsCheck {
/**
 * 操作系统的类型
 */
public enum OSType {
    Windows, MacOS, Linux, Other
};

//缓存操作系统检测的结果
protected static OSType detectedOS;

/**
 *从 os.name 系统属性中检测操作系统并缓存结果
 *
```


```
     * @returns - 检测到的操作系统
     */
    public static OSType getOperatingSystemType() {
        if (detectedOS == null) {
            String OS = System.getProperty("os.name", "generic").
            toLowerCase(
                    Locale.ENGLISH);
            if ((OS.indexOf("mac") >= 0) ||(OS.indexOf("darwin") >= 0)) {
                detectedOS = OSType.MacOS;
            } else if (OS.indexOf("win") >= 0) {
                detectedOS = OSType.Windows;
            } else if (OS.indexOf("nux") >= 0) {
                detectedOS = OSType.Linux;
            } else {
                detectedOS = OSType.Other;
            }
        }
        return detectedOS;
    }
    }
```

使用如下的代码支持多操作系统。

```
OSType osType = OsCheck.getOperatingSystemType(); //检测操作系统
if (osType == OsCheck.OSType.Linux) {
    // Linux 操作系统下加载对应的 so 文件
    libName = "libopencv_java320.so";
} else if (osType == OsCheck.OSType.Windows) {
    // Windows 操作系统下加载对应的 dll 文件
    libName = "opencv_java320.dll";
}
System.load(new File("./lib/".concat(libName)).getAbsolutePath());
```

可以使用 GPU + OpenCL 来提高 OpenCV 的运行速度，也可以使用 JavaCV 稳定地加载 C++实现的动态链接库。如果在 64 位的 JVM 和 Windows 下使用，则在项目中添加对 javacpp.jar、opencv-windows-x64.jar 和 opencv.jar 的引用。JavaCV 检测人脸的代码如下：

```
import org.bytedeco.javacpp.opencv_core.Point;
import org.bytedeco.javacpp.opencv_core.Rect;
import org.bytedeco.javacpp.opencv_core.RectVector;
import org.bytedeco.javacpp.opencv_core.Scalar;
import org.bytedeco.javacpp.opencv_imgcodecs;
import org.bytedeco.javacpp.opencv_core.Mat;
import org.bytedeco.javacpp.opencv_imgproc;
import org.bytedeco.javacpp.opencv_objdetect.CascadeClassifier;

public class App {
    public static void main(String[] args) {
        String filepath = "info1.jpg";          //文件路径不要有斜线或者中文字符
        faceDetect(filepath);                   //调用检测方法
    }

    public static void faceDetect(String filepath) {
        String classifierName = "haarcascade_frontalface_alt.xml";
```

```
                                        //模型文件名
        CascadeClassifier faceDetector = new CascadeClassifier
        (classifierName);               //创建级联分类器
        Mat source = opencv_imgcodecs.imread(filepath);
                                        //将图片读入矩阵对象中
        System.out.println("null? "+source.empty());
                                        //检查是否正确读入
        RectVector faceDetections = new RectVector();
                                        //识别结果放入矩阵数组
        faceDetector.detectMultiScale(source, faceDetections);
                                        //执行人脸检测
        System.out.println(faceDetections.size() + " faces are detected!");
                                        //输出检测出来的人脸数量
        // 遍历识别结果数组
        for (int i = 0; i < faceDetections.size(); i++) {
        Rect r = faceDetections.get(i);     //矩阵
        opencv_imgproc.rectangle(source, new Point(r.x(), r.y()), new
         Point(r.x()
                  + r.width(), r.y() + r.height()), new Scalar(0, 0, 255,
                  0));                  //画出矩形框
        }
        opencv_imgcodecs.imwrite("faces.png", source);
                                        //保存识别出人脸的图像
        faceDetector.close();           //关闭分类器
    }
}
```

5.6 本章小结

为了实现更好的搜索准确性，可以改进检索模型。

检索模型是由 Salton 等人于 20 世纪 70 年代提出的向量空间检索模型发展而来。向量空间检索模型最开始以 TF（词频）作为维度的值，然后发展成以 TF*IDF 作为维度值。

BM25 源自 20 世纪 70 年代伦敦城市大学第一个实现这个函数的系统——Okapi 信息检索系统。它是基于 20 世纪 70～80 年代由 Stephen E. Robertson 和 Karen Spärck Jones 等人开发的概率检索框架。20 世纪 80 年代末概率模型（特别是以 Okapi 系统为代表的 BM25 系列算法）出现并逐渐替代了经典模型在信息检索模型领域的地位，成为新兴的且功能强大、表现越来越出色的模型。

BM25 和 BM25F 两种模型在 TREC 文本检索评测会议中都有很好的表现，并且公认是目前 IR 范围内最先进的检索模型。BM25 适用于没有结构的全文检索，而 BM25F 适用于结构化的文档检索，即适用于有好几个全文搜索列的情况，可以使用相关性反馈技术改进 BM25 模型。

LambdaMART 由微软的 Chris Burges 提出，目前在工业界被广泛使用，包括 Bing、Facebook 等都在实际业务中使用了该算法。

除了用于检索文档，机器学习评分算法还可以用于推荐引擎。例如，购物网站给访问网站的用户推荐商品。

除了 RankLib，还可以使用 XGBoost（网址是 https://github.com/dmlc/xgboost）实现学习评分。

可以根据用户在搜索结果页的行为自动学习排名，调整搜索系统，实现信息的按需提供。

第 6 章　搜索界面开发

搜索相关页面主要包括首页和搜索结果页。如果用户输入的搜索词是空的，则可以显示一个对信息分类导航的页面。首页主要包含搜索条区域。此外还可以包含一些推荐信息及当前热门信息。

可以采用模板引擎输出网页，也可以采用前、后端分离的微服务架构开发搜索界面。微服务架构可以采用 Spring Boot 实现。

6.1　使用 Searchkit 实现搜索界面

Searchkit 是一套 React 构建的 UI 组件，目标是使用声明式的组件帮助用户快速创建漂亮的搜索应用程序，而不是使用户先成为一个 Elasticsearch 专家。

Searchkit 推荐的项目设置使用 Webpack 和 TypeScript，Searchkit 还支持与 ES6（ECMAScript 6）/ Webpack 一起使用。建议通过 npm 安装 Searchkit。命令如下：

```
# yum install npm
```

建议使用 Webpack 进行 Searchkit 的 SRC、CSS 和静态资源的模块依赖管理。需要 SCSS，文件加载器才能正确解决 Searchkit 的依赖关系。命令如下：

```
# npm install searchkit --save
```

如果碰到 OpenSSL 的版本问题，可以运行：

```
# yum update openssl
```

使用 Webpack / ES6 导入相关模块：

```
import {
    SearchBox,                    //搜索框
    Hits,                         //命中结果
    Pagination                    //分页
    } from "searchkit";
```

Searchkit 库脚本可通过 bower 或 jsDelivr CDN 获得。bower 是一个客户端技术的软件包管理器，和 npm 类似。通过 bower 安装 Searchkit 命令如下：

```
# bower install searchkit --save
```

CDN 脚本中相关的 JS 和 CSS 文件如下：

```
<script type="text/javascript" src="//cdn.jsdelivr.net/react/0.14.7/
react.min.js"></script>
<script type="text/javascript"
   src="//cdn.jsdelivr.net/react/0.14.7/react-dom.min.js"></script>
<script type="text/javascript" src="//cdn.jsdelivr.net/searchkit/0.10.0/
bundle.js"></script>
<link rel="stylesheet" type="text/css" href="//cdn.jsdelivr.net/
searchkit/0.10.0/theme.css">
```

下面是将 Searchkit 连接到本地 Elasticsearch 的一个例子。如果碰到一个访问控制（Cors）相关的错误，则需要添加以下内容到 config/elasticsearch.yml 文件中。

```
http.cors.enabled : true
http.cors.allow-origin : "*"
http.cors.allow-methods : OPTIONS, HEAD, GET, POST, PUT, DELETE
http.cors.allow-headers : X-Requested-With,X-Auth-Token,Content-Type,
Content-Length
```

为了使用 Searchkit，需要用一个像主机 URL 一样的 Elasticsearch 去实例化一个 SearchkitManager。然后包装一个 Searchkit 应用程序并呈现给页面。通过配置创建 Searchkit Manager 对象如下：

```
const searchkit =
   new SearchkitManager("http://localhost:9200/")
                                                    //实例化 SearchkitManager

<SearchkitProvider searchkit={searchkit}>          //使用 Searchkit 的标签
...
</SearchkitProvider>
```

Express 是一个简洁而灵活的 node.js Web 应用框架。这里用 node express 构建了一个叫做 searchkit-express 的插件。这个代理插件（Searchkit-express）通过服务器将搜索请求代理到 Elasticsearch 上。这样能验证服务器上的请求，能够在请求到达 Elasticsearch 实例之前使用路由器选项应用额外的过滤器进行过滤。

代理连接的配置如下：

```
const searchkit = new SearchkitManager("/")  //创建 SearchkitManager 对象

<SearchkitProvider searchkit={searchkit}>
...
</SearchkitProvider>
```

使用电影网址实例化一个 SearchkitManager。然后包装一个 Searchkit 应用程序并呈现给页面。

```
import * as React from "react";                    //导入 React 模块
import * as ReactDOM from "react-dom";             //导入 ReactDOM 模块

import {
  SearchkitManager, SearchkitProvider
} from "searchkit";  //导入 SearchkitManager 和 SearchkitProvider 模块
```

```
const searchkit = new SearchkitManager("http://localhost:9200/movies");

//将模板转为 HTML 语言，并插入指定的 DOM 节点
ReactDOM.render((
    <SearchkitProvider searchkit={searchkit}>
        <div>
            <SearchBox/>
            <Hits/>
        </div>
    </SearchkitProvider>
),  document.getElementById('root'))                    //根据 ID 查找 DOM 节点
```

搜索框组件是用户输入搜索查询的地方。例如：

```
import {
  SearchBox,
  SearchkitComponent
} from "searchkit";          //导入SearchkitManager和SearchkitProvider模块

class App extends SearchkitComponent {
  render(){
    <div>
      <SearchBox
        searchOnChange={true}
        queryOptions={{analyzer:"standard"}}      //查询所用的分析器"standard"
        queryFields={["title^5", "languages", "text"]}/>
                                                  //查询列：标题、语言和文本列
    </div>
  }
}
```

SearchBox 组件对外公开的属性列表如下。

- searchOnChange（布尔值）：可选的属性，在用户输入时更新搜索结果，默认为 false。如果和 prefixQueryFields 一起使用，可以在用户输入时获得更好的搜索结果。
- queryBuilder（函数）：用于创建发送给 Elasticsearch 的查询，默认为 SimpleQueryString。
- queryFields（数组）：可选的属性，在其中搜索一组 Elasticsearch 字段，可以指定特定字段的加权，默认搜索为_all。
- queryOptions（对象）：可选的属性，查询字符串的选项对象。
- prefixQueryFields（数组）：可选，在其中搜索一组 Elasticsearch 字段，可以指定特定字段的加权，默认搜索为_all。只有在 searchOnChange 为 true 时才会使用。
- prefixQueryOptions（对象）：可选，MultiMatchQuery 的选项对象。
- mod（字符串）：可选，一个自定义的 BEM（Block-Element-Modifier）容器类。
- translations（对象）：希望覆盖的翻译对象。
- placeholder（字符串）：输入框的占位符。
- searchThrottleTime（数字）：默认是 200 毫秒，当 searchOnChange 属性为 true 时使用。对 Elasticsearch 的搜索只会在每隔 searchThrottleTime 所设定的时间就被调用一次。

- blurAction（搜索|恢复）：当 searchOnChange = {false} 时，配置搜索框失去焦点时的 SearchBox 行为。默认为搜索。

命中组件（即 Hits 组件）显示 Elasticsearch 的结果。要定制每个结果，则需要实现一个 React 组件并传递给 itemComponent 属性。该组件将从搜索结果中收到一个命中对象，其中包含 result._source 属性，以及索引的存储字段。用法示例如下：

```
import { get } from "lodash";       //导入 Lodash 工具库

import {
  Hits,
  SearchkitComponent,
  HitItemProps
} from "searchkit";     //导入 Hits、SearchkitComponent 和 HitItemProps 组件

//定义命中项
const HitItem = (props) => (
  <div
className={props.bemBlocks.item().mix(props.bemBlocks.container("item")
)}>
    <img  className={props.bemBlocks.item("poster")}  src={props.result._
source.poster}/>
    <div className={props.bemBlocks.item("title")} dangerouslySetInnerHTML=
{{__html:
get(props.result,"highlight.title",props.result._source.title)}}></div>
  </div>
)

class App extends SearchkitComponent {

  render(){                          //显示命中结果
    <div>
      //每页显示 50 条记录
      <Hits  hitsPerPage={50}  highlightFields={["title"]}  sourceFilter=
{["title", "poster", "imdbId"]}
      mod="sk-hits-grid" itemComponent={HitItem}/>
    </div>
  }
}
```

Hits 组件对外公开的属性列表如下。

- hitsPerPage（数值）：每页显示的结果数量。
- highlightFields（数组）：高亮显示的字段数组，任何高亮匹配都会在 result.highlight [fieldName]中返回。
- customHighlight（对象）：可选属性，允许任何自定义高亮显示行为来控制片段数量、片段大小和高亮器，作为高亮的值直接传递给 Elasticsearch。
- mod（字符串）：可选属性。一个自定义的 BEM 容器类。
- itemComponent（React 组件）：用于每个命中渲染的 React 组件。
- listComponent（React 组件）：如果要控制结果的完整列表，请使用这个 React 组件。

- sourceFilter（字符串|布尔值|数组）：发送给 Elasticsearch 的源过滤器参数，用于减少结果中的_source 数据。
- scrollTo（字符串|布尔值）：当结果发生变化时，使用 scrollTo 属性中传递的 jQuery 样式选择器会滚动到元素的顶部。如果为 true，则使用 body 作为选择器。默认值为 true。如果为 false，则在渲染新结果时不会滚动。可以通过比较命中的_id 字段和新的结果来确定一个变化。

如果当前查询没有得到 Elasticsearch 的结果时，将显示 NoHits 组件。NoHits 可能会提供帮助用户调整搜索以返回结果的操作。Elasticsearch 响应错误时，NoHits 将显示错误。可以通过在组件或 errorComponent 属性中传入 React 组件来覆盖 NoHits 的显示。

分页组件提供了进入下一页和上一页的功能。示例如下：

```
import {
  Pagination,                                    //导入分页组件
  SearchkitComponent
} from "searchkit";

class App extends SearchkitComponent {

  render(){
    <div>
      <Pagination showNumbers={true}/>           //显示分页链接
    </div>
  }
}
```

6.2 Spring Boot 入门

Spring Boot 应用可以直接由 Maven 的 plugin 打包成 jar 包。在部署时，直接使用 java -jar application.jar 启动 Spring Boot。

首先通过命令行下载 Spring Boot 入门源代码：

```
# git clone https://github.com/spring-guides/gs-spring-boot.git
```

然后切换到完整的例子目录：

```
# cd gs-spring-boot/complete/
```

使用 Maven 构建项目并运行：

```
# mvn package && java -jar target/gs-spring-boot-0.1.0.jar
```

然后在另外一个控制台下查看启动的服务。

```
# http http://localhost:8080/
```

为了加快下载速度，修改 pom.xml 中央仓库地址为：

```
http://maven.aliyun.com/nexus/content/groups/public/。
```

如果是在 Windows 下，则可以用如下命令构建 JAR 文件：

```
>./mvnw clean package
```

然后运行构建出来的这个 JAR 文件：

```
>java -jar target/gs-spring-boot-0.1.0.jar
```

如是在 Eclipse 中开发 Spring Boot 项目，其中 XML 格式的.classpath 文件中的 classpathentry 标签保存了各种类路径信息。

classpathentry 标签的 kind 属性表示类路径的类型。其中，src 类型表示源代码类型的类路径；con 类型表示 classpath 容器。classpathentry 标签的 path 属性表示路径，且使用的路径都是相对于.classpath 路径的。

.classpath 文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
<classpathentry kind="con" path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
<classpathentry kind="con"
        path="org.eclipse.m2e.MAVEN2_CLASSPATH_CONTAINER">
    <attributes>
        <attribute name="maven.pomderived" value="true"/>
    </attributes>
</classpathentry>
<classpathentry kind="src" path="resources"/>
<classpathentry kind="src" path="src"/>
<classpathentry kind="output" path="target/classes"/>
</classpath>
```

在项目中引入依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>1.5.6.RELEASE</version>
</dependency>
```

让首页可以访问 index.html：

```
@Configuration
public class WebMvcConfiguration extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        //控制器注册表中增加视图控制器
        registry.addViewController("/").setViewName("index.html");
        //设置最高的优先级
        registry.setOrder(Ordered.HIGHEST_PRECEDENCE);
    }

}
```

在 IDEA 中开发 Spring Boot 应用，可以新建一个 Maven 项目，然后创建一个控制类：

```
package com.example.demo;
```

```
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
public class HelloController {

    @RequestMapping("/")
    public String index() {
        return "Greetings from Spring Boot!";
    }

}
```

再创建一个 Application 类：

```
package com.example.demo;

import java.util.Arrays;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    public CommandLineRunner commandLineRunner(ApplicationContext ctx) {
        return args -> {

            System.out.println("Let's inspect the beans provided by Spring
Boot:");

            String[] beanNames = ctx.getBeanDefinitionNames();
            Arrays.sort(beanNames);
            for (String beanName : beanNames) {
                System.out.println(beanName);
            }

        };
    }

}
```

静态页面或者 JS、CSS、图片之类的静态资源可以放入 resources 目录的 public 目录下。

6.2.1　可执行的 WAR

Spring Boot Web 应用程序可以打包在 JAR 文件中，也可以打包在 WAR 文件中。在这两个选项中，嵌入的 Servlet 容器（默认为 Tomcat）也可以包含在包中，以便将归档文件（JAR 或 WAR）作为独立的应用程序来执行。只有当我们使用 spring-boot maven 插件时，才可能实现归档文件独立执行的功能。

下面首先创建一个传统的 Maven WAR 文件并将其部署到 Tomcat 服务器上，然后使用 spring-boot 插件创建一个可执行 WAR 文件，并将其部署到服务器上并执行。我们将讲解传统的 WAR 文件与可执行 WAR 文件在创建和使用上的区别，并将分析由 spring-boot maven 插件创建的 WAR 文件的结构。

下面创建一个以 JSP 作为展示层的 Web 示例项目。

```
@SpringBootApplication
public class WarStructureExample extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure (SpringApplicationBuilder
    builder) {
        return builder.sources(WarStructureExample.class);
    }

    public static void main (String[] args) {
        SpringApplication.run(WarStructureExample.class, args);
                                                    //设置 SpringApplication 的源
    }

    @Controller
    public static class MyController {

            @RequestMapping("/")
            public String handler (Model model) {
            model.addAttribute("msg",
                            "a war structure example");
            return "myPage";
        }
    }
}
```

在上面的代码中，主类 WarStructureExample 扩展 org.springframework.boot.web. support. SpringBootServletInitializer，该类进而扩展 WebApplicationInitializer 接口。WebApplication Initializer 接口基于 Servlet 3.0 ServletContainerInitializer 概念。

这个扩展的目的是通过 WebApplicationInitializer 接口设置 Servlet 上下文。另外，它要求子类设置 SpringApplication 的源（使用@SpringBootApplication 类注解的类），以便它可以用一个有效的源调用 SpringApplication.run()，进行自动配置和应用程序级别的 Bean 连接等。这种安排只在应用程序作为 WAR 文件部署在 Servlet 容器中时才需要。在 Web

容器中，main()方法不能像在独立的应用程序或可执行 JAR 或 WAR 文件中那样执行。

src/main/webapp/WEB-INF/pages/myPage.jsp 内容如下：

```
<h2>From JSP page </h2>
<%@ page language="java"
    contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<html>
<body>
    Message : ${msg}
</body>
</html>
```

配置文件 src/main/resources/application.properties 中的内容如下：

```
spring.mvc.view.prefix= /WEB-INF/pages/
spring.mvc.view.suffix= .jsp
```

如果想通过 Maven 普通包的目标而不是通过 Spring Boot 插件创建一个 WAR 文件，那么就不需要 Spring Boot 插件。这个 WAR 文件当然是不可执行的，但可以部署到一个运行 Servlet 容器的服务器上。

pom.xml 文件内容如下：

```
<project ....>
 <modelVersion>4.0.0</modelVersion>

 <groupId>com.lietu.example</groupId>
 <artifactId>traditional-war</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>war</packaging>

 <parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.3.RELEASE</version>
 </parent>

 <properties>
  <java.version>1.8</java.version>
 </properties>

 <dependencies>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-tomcat</artifactId>
   <scope>provided</scope>
  </dependency>
 </dependencies>
</project>
```

spring-boot-starter-parent 项目的 pom.xml 文件默认配置使用 maven-war-plugin，所以

在 pox.xml 中只需要设置 packaging 的值为 war 即可。

此外，随 spring-boot-starter-web 项目一起提供的嵌入式 Servlet 容器可能会干扰将部署 WAR 文件的 Servlet 容器。为了避免这种情况，需要将默认的依赖范围重写为 provided，这样最终的 WAR 文件就不会包含嵌入的 Tomcat 服务器。

打包 WAR 文件。首先查看打包前的文件目录结构：

```
D:\examples\traditional-war>mvn -q clean
```

然后使用 tree 命令查看文件目录结构：

```
D:\examples\traditional-war>tree /A /F
Folder PATH listing for volume Data
Volume serial number is 00000051 68F9:EDFA
D:.
|   pom.xml
|
\---src
    \---main
        +---java
        |   \---com
        |       \---logicbig
        |           \---example
        |                   WarStructureExample.java
        |
        +---resources
        |       application.properties
        |
        \---webapp
            \---WEB-INF
                \---pages
                        myPage.jsp
```

执行打包：

```
D:\examples\traditional-war>mvn -q package
```

然后查看文件目录结构在打包后的变化：

```
D:\examples\traditional-war>tree /A /F
Folder PATH listing for volume Data
Volume serial number is 00000024 68F9:EDFA
D:.
|   pom.xml
|
+---src
|   \---main
|       +---java
|       |   \---com
|       |       \---logicbig
|       |           \---example
|       |                   WarStructureExample.java
|       |
|       +---resources
|       |       application.properties
|       |
```

```
|       \---webapp
|           \---WEB-INF
|               \---pages
|                       myPage.jsp
|
\---target
    |   traditional-war-1.0-SNAPSHOT.war
    |
    +---classes
    |   |   application.properties
    |   |
    |   \---com
    |       \---logicbig
    |           \---example
    |                   WarStructureExample$MyController.class
    |                   WarStructureExample.class
    |
    +---generated-sources
    |   \---annotations
    +---maven-archiver
    |       pom.properties
    |
    +---maven-status
    |   \---maven-compiler-plugin
    |       \---compile
    |           \---default-compile
    |                   createdFiles.lst
    |                   inputFiles.lst
    |
    \---traditional-war-1.0-SNAPSHOT
        +---META-INF
        \---WEB-INF
            +---classes
            |   |   application.properties
            |   |
            |   \---com
            |       \---logicbig
            |           \---example
            |                   WarStructureExample$MyController.class
            |                   WarStructureExample.class
            |
            +---lib
            |       classmate-1.3.3.jar
            |       hibernate-validator-5.2.4.Final.jar
            |       jackson-annotations-2.8.5.jar
            |       jackson-core-2.8.5.jar
            |       jackson-databind-2.8.5.jar
            |       jboss-logging-3.3.0.Final.jar
            |       jcl-over-slf4j-1.7.22.jar
            |       jul-to-slf4j-1.7.22.jar
            |       log4j-over-slf4j-1.7.22.jar
            |       logback-classic-1.1.8.jar
            |       logback-core-1.1.8.jar
            |       slf4j-api-1.7.22.jar
            |       snakeyaml-1.17.jar
```

```
|       spring-aop-4.3.5.RELEASE.jar
|       spring-beans-4.3.5.RELEASE.jar
|       spring-boot-1.4.3.RELEASE.jar
|       spring-boot-autoconfigure-1.4.3.RELEASE.jar
|       spring-boot-starter-1.4.3.RELEASE.jar
|       spring-boot-starter-logging-1.4.3.RELEASE.jar
|       spring-boot-starter-web-1.4.3.RELEASE.jar
|       spring-context-4.3.5.RELEASE.jar
|       spring-core-4.3.5.RELEASE.jar
|       spring-expression-4.3.5.RELEASE.jar
|       spring-web-4.3.5.RELEASE.jar
|       spring-webmvc-4.3.5.RELEASE.jar
|       validation-api-1.1.0.Final.jar
|
\---pages
        myPage.jsp
```

在 Tomcat 服务器上部署 WAR：

（1）单击<tomcat-home> /bin/startup.bat 文件，启动 Tomcat 服务器。

（2）进入 localhost：8080 的 Tomcat 主页面，单击 Manager App 按钮。如果要求输入用户名和密码，则输入相应的用户名和密码。现在到达“Tomcat web 应用程序管理器”页面。

（3）在“部署 WAR 文件”部分（接近底部）下“选择文件”，此时必须选择在项目目标文件夹下创建的 WAR 文件。单击“部署”按钮，将 WAR 文件部署到服务器上。

（4）在 Applications 表中（靠近顶部）找到“/traditional-war-1.0-SNAPSHOT”并单击，会看到 JSP 模板显示出的输出。

使用 Spring Boot 插件打包 WAR。将使用与上面相同的主类，在 pom.xml 文件中会有一些重要的改变。这里要添加 Spring Boot Maven 插件，代码如下：

```
<project ..........>
 <modelVersion>4.0.0</modelVersion>

 <groupId>com.lietu.example</groupId>
 <artifactId>boot-war</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>war</packaging>

 <parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.3.RELEASE</version>
 </parent>

 <properties>
  <java.version>1.8</java.version>
 </properties>
 <dependencies>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
```

```
  </dependency>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-tomcat</artifactId>
   <scope>provided</scope>
  </dependency>
  <dependency>
   <groupId>org.apache.tomcat.embed</groupId>
   <artifactId>tomcat-embed-jasper</artifactId>
   <scope>provided</scope>
  </dependency>
 </dependencies>

 <build>
  <plugins>
   <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>1.4.3.RELEASE</version>
    <executions>
     <execution>
      <goals>
       <goal>repackage</goal>
      </goals>
     </execution>
    </executions>
   </plugin>
  </plugins>
 </build>
</project>
```

请注意，还需要添加额外的 tomcat-embed-jasper（一个 Core Tomcat 实现）依赖项，这个只有在运行可执行文件 WAR 的时候才需要。

打包 WAR 文件：

```
D:\examples\boot-war>mvn -q clean

D:\examples\boot-war>tree /A /F
Folder PATH listing for volume Data
Volume serial number is 000000F6 68F9:EDFA
D:.
|   pom.xml
|
\---src
    \---main
        +---java
        |   \---com
        |       \---logicbig
        |           \---example
        |                   WarStructureExample.java
        |
        +---resources
        |       application.properties
        |
        \---webapp
```

```
            \---WEB-INF
               \---pages
                     myPage.jsp

D:\examples\boot-war>mvn -q package

D:\examples\boot-war>tree /A /F
Folder PATH listing for volume Data
Volume serial number is 00000074 68F9:EDFA
D:.
|   pom.xml
|
+---src
|   \---main
|       +---java
|       |   \---com
|       |       \---logicbig
|       |           \---example
|       |                   WarStructureExample.java
|       |
|       +---resources
|       |       application.properties
|       |
|       \---webapp
|           \---WEB-INF
|               \---pages
|                     myPage.jsp
|
\---target
    |   boot-war-1.0-SNAPSHOT.war
    |   boot-war-1.0-SNAPSHOT.war.original
    |
    +---boot-war-1.0-SNAPSHOT
    |   +---META-INF
    |   \---WEB-INF
    |       +---classes
    |       |   |   application.properties
    |       |   |
    |       |   \---com
    |       |       \---logicbig
    |       |           \---example
    |       |                   WarStructureExample$MyController.class
    |       |                   WarStructureExample.class
    |       |
    |       +---lib
    |       |       classmate-1.3.3.jar
    |       |       hibernate-validator-5.2.4.Final.jar
    |       |       jackson-annotations-2.8.5.jar
    |       |       jackson-core-2.8.5.jar
    |       |       jackson-databind-2.8.5.jar
    |       |       jboss-logging-3.3.0.Final.jar
    |       |       jcl-over-slf4j-1.7.22.jar
    |       |       jul-to-slf4j-1.7.22.jar
    |       |       log4j-over-slf4j-1.7.22.jar
    |       |       logback-classic-1.1.8.jar
```

```
|       |       logback-core-1.1.8.jar
|       |       slf4j-api-1.7.22.jar
|       |       snakeyaml-1.17.jar
|       |       spring-aop-4.3.5.RELEASE.jar
|       |       spring-beans-4.3.5.RELEASE.jar
|       |       spring-boot-1.4.3.RELEASE.jar
|       |       spring-boot-autoconfigure-1.4.3.RELEASE.jar
|       |       spring-boot-starter-1.4.3.RELEASE.jar
|       |       spring-boot-starter-logging-1.4.3.RELEASE.jar
|       |       spring-boot-starter-web-1.4.3.RELEASE.jar
|       |       spring-context-4.3.5.RELEASE.jar
|       |       spring-core-4.3.5.RELEASE.jar
|       |       spring-expression-4.3.5.RELEASE.jar
|       |       spring-web-4.3.5.RELEASE.jar
|       |       spring-webmvc-4.3.5.RELEASE.jar
|       |       validation-api-1.1.0.Final.jar
|       |
|       \---pages
|               myPage.jsp
|
+---classes
|   |   application.properties
|   |
|   \---com
|       \---logicbig
|           \---example
|                   WarStructureExample$MyController.class
|                   WarStructureExample.class
|
+---generated-sources
|   \---annotations
+---maven-archiver
|       pom.properties
|
\---maven-status
    \---maven-compiler-plugin
        \---compile
            \---default-compile
                    createdFiles.lst
                    inputFiles.lst

D:\examples\boot-war>
```

boot-war-1.0-SNAPSHOT 文件夹下的内容是原始的 WAR 文件内容。

要查看 spring-boot 插件增强的 WAR 文件，必须提取 boot-war-1.0-SNAPSHOT.war 文件。代码如下：

```
D:\examples\boot-war>md temp

D:\examples\boot-war>cd temp

D:\examples\boot-war\temp>jar -xf ..\target\boot-war-1.0-SNAPSHOT.war

D:\examples\boot-war\temp>tree /A /F
Folder PATH listing for volume Data
```

```
Volume serial number is 000000B9 68F9:EDFA
D:.
+---META-INF
|   |   MANIFEST.MF
|   |
|   \---maven
|       \---com.lietu.example
|           \---boot-war
|                   pom.properties
|                   pom.xml
|
+---org
|   \---springframework
|       \---boot
|           \---loader
|               |   ExecutableArchiveLauncher$1.class
|               |   ExecutableArchiveLauncher.class
|               |   JarLauncher.class
|               |   LaunchedURLClassLoader$1.class
|               |   LaunchedURLClassLoader.class
|               |   Launcher.class
|               |   MainMethodRunner.class
|               |   PropertiesLauncher$1.class
|               |   PropertiesLauncher$ArchiveEntryFilter.class
|               |   PropertiesLauncher$FilteredArchive$1.class
|               |   PropertiesLauncher$FilteredArchive.class
|               |   PropertiesLauncher$PrefixMatchingArchiveFilter.class
|               |   PropertiesLauncher.class
|               |   WarLauncher.class
|               |
|               +---archive
|               |       Archive$Entry.class
|               |       Archive$EntryFilter.class
|               |       Archive.class
|               |       ExplodedArchive$1.class
|               |       ExplodedArchive$FileEntry.class
|               |       ExplodedArchive$FileEntryIterator$EntryComparator.
|                       class
|               |       ExplodedArchive$FileEntryIterator.class
|               |       ExplodedArchive.class
|               |       JarFileArchive$EntryIterator.class
|               |       JarFileArchive$JarFileEntry.class
|               |       JarFileArchive.class
|               |
|               +---data
|               |       ByteArrayRandomAccessData.class
|               |       RandomAccessData$ResourceAccess.class
|               |       RandomAccessData.class
|               |       RandomAccessDataFile$DataInputStream.class
|               |       RandomAccessDataFile$FilePool.class
|               |       RandomAccessDataFile.class
|               |
|               +---jar
|               |       AsciiBytes.class
|               |       Bytes.class
```

```
|             |       CentralDirectoryEndRecord.class
|             |       CentralDirectoryFileHeader.class
|             |       CentralDirectoryParser.class
|             |       CentralDirectoryVisitor.class
|             |       FileHeader.class
|             |       Handler.class
|             |       JarEntry.class
|             |       JarEntryFilter.class
|             |       JarFile$1.class
|             |       JarFile$2.class
|             |       JarFile$3.class
|             |       JarFile$JarFileType.class
|             |       JarFile.class
|             |       JarFileEntries$1.class
|             |       JarFileEntries$EntryIterator.class
|             |       JarFileEntries.class
|             |       JarURLConnection$1.class
|             |       JarURLConnection$JarEntryName.class
|             |       JarURLConnection.class
|             |       ZipInflaterInputStream.class
|             |
|             \---util
|                     SystemPropertyUtils.class
|
\---WEB-INF
    +---classes
    |   |   application.properties
    |   |
    |   \---com
    |       \---logicbig
    |           \---example
    |                   WarStructureExample$MyController.class
    |                   WarStructureExample.class
    |
    +---lib
    |       classmate-1.3.3.jar
    |       hibernate-validator-5.2.4.Final.jar
    |       jackson-annotations-2.8.5.jar
    |       jackson-core-2.8.5.jar
    |       jackson-databind-2.8.5.jar
    |       jboss-logging-3.3.0.Final.jar
    |       jcl-over-slf4j-1.7.22.jar
    |       jul-to-slf4j-1.7.22.jar
    |       log4j-over-slf4j-1.7.22.jar
    |       logback-classic-1.1.8.jar
    |       logback-core-1.1.8.jar
    |       slf4j-api-1.7.22.jar
    |       snakeyaml-1.17.jar
    |       spring-aop-4.3.5.RELEASE.jar
    |       spring-beans-4.3.5.RELEASE.jar
    |       spring-boot-1.4.3.RELEASE.jar
    |       spring-boot-autoconfigure-1.4.3.RELEASE.jar
    |       spring-boot-starter-1.4.3.RELEASE.jar
    |       spring-boot-starter-logging-1.4.3.RELEASE.jar
    |       spring-boot-starter-web-1.4.3.RELEASE.jar
```

```
|       spring-context-4.3.5.RELEASE.jar
|       spring-core-4.3.5.RELEASE.jar
|       spring-expression-4.3.5.RELEASE.jar
|       spring-web-4.3.5.RELEASE.jar
|       spring-webmvc-4.3.5.RELEASE.jar
|       validation-api-1.1.0.Final.jar
|
+---lib-provided
|       ecj-4.5.1.jar
|       spring-boot-starter-tomcat-1.4.3.RELEASE.jar
|       tomcat-embed-core-8.5.6.jar
|       tomcat-embed-el-8.5.6.jar
|       tomcat-embed-jasper-8.5.6.jar
|       tomcat-embed-websocket-8.5.6.jar
|
\---pages
        myPage.jsp

D:\examples\boot-war\temp>
```

在上面的输出中，下面的结果是显而易见的。

- 'provided'依赖放置在'WEB-INF/lib-provided'中。这是为了避免 WAR 部署在 Servlet 容器中时发生任何冲突。
- 只有当作为应用程序执行 WAR 文件并使用嵌入式服务器时，才会使用 'WEB-INF/lib-provided' jar。通常在命令行上使用 java –jar 将 WAR 文件作为应用程序来运行。

有一个特殊的文件夹：org/springframework/boot/loader/，其中包含从 WAR 文件启动可执行应用程序的类。当作为可执行的 WAR 文件运行时，将使用 WarLauncher。让我们在 META-INF/Manifest.MF 文件中确认这一点。

```
Manifest-Version: 1.0
Implementation-Title: boot-war
Implementation-Version: 1.0-SNAPSHOT
Archiver-Version: Plexus Archiver
Built-By: Joe
Implementation-Vendor-Id: com.lietu.example
Spring-Boot-Version: 1.4.3.RELEASE
Implementation-Vendor: Lietu Software, Inc.
Main-Class: org.springframework.boot.loader.WarLauncher
Start-Class: com.lietu.example.WarStructureExample
Spring-Boot-Classes: WEB-INF/classes/
Spring-Boot-Lib: WEB-INF/lib/
Created-By: Apache Maven 3.0.5
Build-Jdk: 1.8.0_111
Implementation-URL: http://projects.spring.io/spring-boot/boot-war/
```

由于启动器类和嵌入式服务器 JAR 文件相关的所有文件只包含在 lib-provided 文件夹中，所以引导 WAR 扩展只是为了 WAR 的可执行性，并且在部署到 Servlet 容器中时不会发挥任何作用。

部署这个 WAR 文件到 Tomcat 服务器的方法，与传统的 WAR 文件方法相同。

6.2.2 spring-boot-devtools 模块实现热部署

Spring Boot 在 1.3 版本中引入了 spring-boot-devtools 模块。这个模块的主要目标是通过在项目的开发阶段启用一些重要的关键特性来提高开发速度。

使用 spring-boot-devtools 模块的应用程序将在类路径上的文件发生更改时自动重启。在 IDE 中工作时，这是一个非常有用的功能，因为它为代码更改提供了一个非常快速的反馈循环。

默认情况下系统会监视类路径上的所有变动。但请注意，某些资源（如静态资源和视图模板）不需要重启应用程序。

想要使用 devtools 支持，只需将模块依赖关系添加到构建中。对于 Maven 构建，添加如下依赖：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

重启功能对于使用标准 ObjectInputStream 实现反序列化的对象会出错。例如，使用序列化和反序列化将 Java 对象存储到 Redis 数据库中时，可能会出现类型转换异常。如果需要反序列化数据，可能需要使用 Spring 的 ConfigurableObjectInputStream 配合 Thread.currentThread().getContextClassLoader()使用。

6.3 Java 模板引擎 Pebble 介绍

可以使用模板引擎 Pebble 实现搜索首页。首先，将以下依赖项添加到项目的 pom.xml 文件中：

```
<dependency>
<groupId>com.mitchellbosecke</groupId>
<artifactId>pebble</artifactId>
<version>2.4.0</version>
</dependency>
```

创建一个网页模板，home.html 文件内容如下：

```
<html>
<head>
    <title>{{ websiteTitle }}</title>
</head>
<body>
    {{ content }}
```

```
</body>
</html>
```

用代码生成网页：

```
PebbleEngine engine = new PebbleEngine.Builder().build();
                                                    //构建出 PebbleEngine
//根据文件得到模板
PebbleTemplate compiledTemplate = engine.getTemplate("home.html");
//把模板生成的网页写入字符串中
Writer writer = new StringWriter();

//通过键/值对传入模板参数
Map<String, Object> context = new HashMap<>();
context.put("websiteTitle", "My First Website");
context.put("content", "My Interesting Content");

//模板结合模板参数
compiledTemplate.evaluate(writer, context);

String output = writer.toString();                  //得到生成的网页结果
System.out.println(output);
```

PebbleEngineBuilder 也能接受一个加载器（Loader）实现作为参数。加载器加载程序负责查找模板。Pebble 自带如下加载器实现。

- ClasspathLoader：使用类加载器来搜索当前的类路径。
- FileLoader：使用文件系统路径查找模板。
- ServletLoader：使用 Servlet 上下文来查找模板。
- StringLoader：直接提供模板的内容。
- DelegatingLoader：一个子加载器集合的代理。

下面是使用 StringLoader 的例子。

```
PebbleEngine engine = new PebbleEngine.Builder().loader(
    new StringLoader()).build();    //使用 StringLoader 构建出 PebbleEngine

StringWriter writer = new StringWriter();

Map<String, Object> context = new HashMap<>();
context.put("name", "lietusearch");
engine.getTemplate("<p>{{name}}</p>").evaluate(writer, context);
                                    //根据字符串得到模板
String result = writer.toString(); //得到结果<p>lietusearch</p>
```

在 Servlet 环境中使用 ServletLoader：

```
public class SearchServlet extends HttpServlet {
@Override
public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
    String qStr = req.getParameter("q");

    ServletOutputStream out = resp.getOutputStream();
```

```
        resp.setContentType("text/html;charset=UTF-8");
        try {
            ServletContext sc = getServletContext();
            PebbleEngine engine = new PebbleEngine.Builder().loader(
                    new ServletLoader(sc)).build();
            PebbleTemplate compiledTemplate =
                    engine.getTemplate("searchResult.html");
            compiledTemplate.getName();
            Writer writer = new StringWriter();

            Map<String, Object> context = new HashMap<>();
            context.put("websiteTitle", qStr);
            context.put("content", "search Content");

            compiledTemplate.evaluate(writer, context);

            String result = writer.toString();
            out.println(result);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

可以使用 Mockito 测试这个 Servlet 类。先使用 Mockito 和 JUnit 进行测试：

```
public class TestMock extends TestCase {
public static void testList() {
    List mock = Mockito.mock(List.class);              //创建出指定类型的对象
    System.out.println(" mock.get(0):" + mock.get(0));      //返回 null
    Mockito.when(mock.get(0)).thenReturn(1);           //设置对象的行为
    System.out.println(" mock.get(0):" + mock.get(0));      //返回 1
}
}
```

Mockito 使用 Byte Buddy 生成 Java 字节码来创造出需要的对象。

测试 Servlet 用的存根类 FakeServletOutputStream 代码如下：

```
public class FakeServletOutputStream extends ServletOutputStream {
public ByteArrayOutputStream baos = new ByteArrayOutputStream();

public void write(int i) throws IOException {
    baos.write(i);
}

public String getContent() {
    return baos.toString();                            //返回输出的内容
}

@Override
public boolean isReady() {
    return false;
}

@Override
```

```
public void setWriteListener(WriteListener arg0) {
}
}
```

然后结合存根类使用 Mockito 测试 Servlet 类：

```
HttpServletRequest httpServletRequest = Mockito
    .mock(HttpServletRequest.class);                    //得到请求类
Mockito.when(httpServletRequest.getParameter("q")).thenReturn(
    "DNA");                                             //模拟请求类的行为
final HttpServletResponse httpServletResponse = Mockito
    .mock(HttpServletResponse.class);                   //得到响应类

//创建存根类
final FakeServletOutputStream servletOutputStream = new
FakeServletOutputStream();
Mockito.when(httpServletResponse.getOutputStream()).thenReturn(
    servletOutputStream);                               //在响应类中使用存根类

//得到 Servlet 上下文类
final ServletContext servletContext = Mockito.mock(ServletContext.class);

//指定返回搜索首页的本地文件路径
Mockito.when(servletContext.getResourceAsStream("home.html")).
thenReturn(
    new FileInputStream("F:/workspace/PebblePlay/src/home.html"));

//创建用于搜索的 Servlet 实例
SearchServlet searchServlet = new SearchServlet() {
public ServletContext getServletContext() {
    return servletContext;                              //返回 mock
}
};

//设置 Servlet 实例的请求和响应类
searchServlet.doGet(httpServletRequest, httpServletResponse);
System.out.println("content:" + servletOutputStream.getContent());
```

可以通过循环在首页显示一些热门查询词，包含循环的模板：

```
PebbleEngine engine = new PebbleEngine.Builder().loader(
    new StringLoader()).build();                    //创建模板引擎

StringWriter writer = new StringWriter();

//查询项目列表
List<String> searchItems = new ArrayList<>();
searchItems.add("search item1");
searchItems.add("search item2");
searchItems.add("search item3");

Map<String, Object> context = new HashMap<>();
context.put("searchItems", searchItems);
```

```
//设置字符串模板
engine.getTemplate(
        "{% if searchItems is iterable %}{% for searchItem in searchItems
        %}" +
                " \"{{ searchItem }}\" this\n" +
                "{% endfor %}{% else %}nope{% endif %}").evaluate(writer,
                context);
//得到结果
String result = writer.toString();
System.out.println(result);
```

可以把 Pebble 模板引擎作为展示层整合进 SpringBoot 的 MVC 架构中。ViewResolver 是 SpringBoot MVC 的核心组件。将@Controller 中的视图名称转换为实际的 View 实现。ViewResolver 将视图名称映射到实际的视图。请注意，ViewResolver 主要用于 UI 应用程序，而不是 REST 风格的服务（View 不用于呈现@ResponseBody）。

这里专注于 ViewResolver 的配置并启动运行。首先，需要正确配置 ViewResolver 组件。整合 Pebble 模板引擎稍微复杂一些，因为配置的结果会非常紧密地连接到 Servlet 配置本身。所以，回到@BeanMvcConfiguration 并添加：

```
@Bean(name="pebbleViewResolver")    //使用 Bean 注解注册 pebbleViewResolver
public ViewResolver getPebbleViewResolver(){
   PebbleViewResolver resolver = new PebbleViewResolver(); //视图实现类
   resolver.setPrefix("/pebble/");                          //设置前缀

   resolver.setSuffix(".html");                             //设置后缀

   resolver.setPebbleEngine(pebbleEngine());                //设置模板引擎

   return resolver;

}
```

模板可以通过 application.properties 进行配置，但是目前 Pebble 模板引擎并不支持该方法，需要通过手动配置，定义更多的 Pebble 相关的@Beans。代码如下：

```
@Bean
public PebbleEngine pebbleEngine() {

  return new PebbleEngine.Builder()
               .loader(this.templatePebbleLoader())
               .extension(pebbleSpringExtension())
               .build();                                    //构建模板引擎

}
```

模板加载器：

```
@Bean
public Loader templatePebbleLoader(){
    return new ServletLoader(servletContext);          //创建 Servlet 加载器
}
```

Spring 扩展 Bean：

```
@Bean
public SpringExtension pebbleSpringExtension() {
  return new SpringExtension();
}
```

返回模板加载器的 templatePebbleLoader()方法需要直接访问 ServletContext，该 ServletContext 需要注入配置 Bean 注解。代码如下：

```
@Autowired
private ServletContext servletContext;
…
```

这样做了以后，Pebble 将接管创建的 Servlet。现在我们已经准备好 Pebble 配置，然后在 webapp 下创建一个新的 pebble 文件夹，并添加一个新的模板文件 pebble.html。代码如下：

```
<html>
<head>
   <title>{{ pebble }}</title>
</head>
<body>
{{ pebble }}
</body>
</html>
```

把模板引擎分配给自己的控制器，负责正确输出生成，代码如下：

```
@Controller
public class PebbleHelloController {
   @RequestMapping(value = "/pebble")          //把请求映射到 pebble 文件夹
   public String something(Model model){          //根据输入的模型输出响应
      System.out.println("Pebble");
      model.addAttribute("pebble", "The Pebble"); //设置模型中的属性值
      return "pebble";
   }
}
```

6.4　通过 Spring-data-elasticsearch 项目访问 Elasticsearch

Spring Data 项目中（下载地址为 https://github.com/spring-projects/spring-data- elasticsearch）提供了一套数据访问层（DAO）的解决方案，致力于减少数据访问层的开发量。Spring-data-elasticsearch 是 Spring Data 的子项目，实现了 Spring Data 访问 Elasticsearch 存储并提供了 Spring Data JPA（Java 持久化 API）模型的访问方式。

Spring Data 项目使用一个叫做 Repository 的接口类为基础。Repository 接口定义如下：

```
public interface Repository<T, ID extends Serializable> {
}
```

Repository 接口是访问底层数据模型的超级接口，而对某种具体的数据访问操作，则在其子接口中定义。例如，Spring-data-elasticsearch 项目中定义了访问 Elasticsearch 数据的 ElasticsearchRepository 接口。ElasticsearchRepository 接口扩展了 PagingAndSortingRepository 接口，它为分页和排序提供了内置的支持。

可以使用 JavaPoet 生成 PagingAndSortingRepository 接口的扩展代码。要指定一个接口，可以使用 TypeSpec.Builder.addSuperinterface()方法。代码如下：

```
TypeSpec userRepository = TypeSpec
    .interfaceBuilder("UserRepository")          //定义接口
    .addAnnotation(Repository.class)             //增加注解
    .addModifiers(Modifier.PUBLIC)               //增加修饰符
    .addSuperinterface(                          //定义接口的父类
        ParameterizedTypeName.get(
                ClassName.get(PagingAndSortingRepository.class),
                ClassName.get(User.class),
                ClassName.get(Long.class))).build();

JavaFile javaFile = JavaFile.builder("com.lietu.helloworld",
    userRepository).build();

javaFile.writeTo(System.out);
```

生成的代码如下：

```
import java.lang.Long;
import
org.springframework.boot.autoconfigure.security.SecurityProperties;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.repository.Repository;

@Repository
public interface UserRepository extends
     PagingAndSortingRepository<SecurityProperties.User, Long> {
}
```

所有继承 Repository 接口的界面都由 Spring 管理，该接口作为标识接口，功能就是用来控制领域模型。Spring Data 项目可以让我们只定义接口，只要遵循 Spring Data 项目的规范，就无须再写实现类。Spring 可以根据接口中定义的方法名实现 Repository 接口。

引入包 spring-data-elasticsearch，并且去掉依赖项 elasticsearch，代码如下：

```
<dependency>
   <groupId>org.springframework.data</groupId>
   <artifactId>spring-data-elasticsearch</artifactId>
   <version>${spring-data-elasticsearch-version}</version>
   <exclusions>
      <exclusion>
         <artifactId>elasticsearch</artifactId>
         <groupId>org.elasticsearch</groupId>
      </exclusion>
   </exclusions>
</dependency>
```

在 XML 文件中配置连接参数，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:elasticsearch="http://www.springframework.org/schema/data/
    elasticsearch"
    xsi:schemaLocation="http://www.springframework.org/schema/data/
    elasticsearch
http://www.springframework.org/schema/data/elasticsearch/spring-
elasticsearch-1.0.xsd
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">
<!--连接参数-->
<elasticsearch:transport-client id="client"
    cluster-nodes="127.0.0.1:9300" cluster-name="lietu" />
<bean name="elasticsearchTemplate"
    class="org.springframework.data.elasticsearch.core.
    ElasticsearchTemplate">
        <constructor-arg name="client" ref="client"/>
    </bean>

    <elasticsearch:repositories
base-package="org.springframework.data.elasticsearch.repositories" />

</beans>
```

为自定义方法扩展 ElasticsearchRepository：

```
public interface BookRepository extends Repository<Book, String> {
    //通过名字和价格查找图书
    List<Book> findByNameAndPrice(String name, Integer price);
    //通过名字或者价格查找图书
    List<Book> findByNameOrPrice(String name, Integer price);
    //通过名字查找图书
    Page<Book> findByName(String name,Pageable page);
    //通过价格区间查找图书
    Page<Book> findByPriceBetween(int price,Pageable page);
    //通过名字模糊查找图书
    Page<Book> findByNameLike(String name,Pageable page);
    //通过内容查找图书
    @Query("{\"bool\" : {\"must\" : {\"term\" : {\"message\" : \"?0\"}}}}")
    Page<Book> findByMessage(String message, Pageable pageable);
}
```

使用 Repository 接口索引单个文档，代码如下：

```
@Autowired
private SampleElasticsearchRepository repository;  //Elasticsearch 存储库

String documentId = "123456";                       //文档编号
SampleEntity sampleEntity = new SampleEntity();     //创建实体类
sampleEntity.setId(documentId);                     //设置文档编号
sampleEntity.setMessage("some message");            //设置消息
```

```
repository.save(sampleEntity);                    //保存实体类到存储库
```

使用 Repository 索引多个文档（批量索引）：

```
@Autowired
private SampleElasticsearchRepository repository;

//创建两个文档
String documentId = "123456";
SampleEntity sampleEntity1 = new SampleEntity();
sampleEntity1.setId(documentId);
sampleEntity1.setMessage("some message");

String documentId2 = "123457"
SampleEntity sampleEntity2 = new SampleEntity();
sampleEntity2.setId(documentId2);
sampleEntity2.setMessage("test message");

//实体类数组
List<SampleEntity> sampleEntities = Arrays.asList(sampleEntity1,
sampleEntity2);

//批量索引
repository.save(sampleEntities);
```

ElasticsearchTemplate 是 Elasticsearch 操作的核心支持类。使用 ElasticsearchTemplate 类索引单个文档，代码如下：

```
//创建单个文档
String documentId = "123456";
SampleEntity sampleEntity = new SampleEntity();
sampleEntity.setId(documentId);
sampleEntity.setMessage("some message");
IndexQuery indexQuery =
new IndexQueryBuilder().withId(sampleEntity.getId()).withObject
(sampleEntity).build();
elasticsearchTemplate.index(indexQuery);
```

使用 ElasticsearchTemplate 类索引多个文档（批量索引），代码如下：

```
@Autowired
private ElasticsearchTemplate elasticsearchTemplate;

List<IndexQuery> indexQueries = new ArrayList<IndexQuery>();
//第一个文档
String documentId = "123456";
SampleEntity sampleEntity1 = new SampleEntity();
sampleEntity1.setId(documentId);
sampleEntity1.setMessage("some message");

IndexQuery indexQuery1 =
      new IndexQueryBuilder().withId(sampleEntity1.getId())
         .withObject(sampleEntity1).build();
indexQueries.add(indexQuery1);

//第二个文档
```

```
String documentId2 = "123457";
SampleEntity sampleEntity2 = new SampleEntity();
sampleEntity2.setId(documentId2);
sampleEntity2.setMessage("some message");

//构建索引查询
IndexQuery indexQuery2 =
    new IndexQueryBuilder().withId(sampleEntity2.getId())
        .withObject(sampleEntity2).build()
indexQueries.add(indexQuery2);

//批量索引
elasticsearchTemplate.bulkIndex(indexQueries);
```

使用 Elasticsearch Template 搜索实体：

```
@Autowired
private ElasticsearchTemplate elasticsearchTemplate;

//构建查询类
SearchQuery searchQuery = new NativeSearchQueryBuilder()
    .withQuery(queryString(documentId).field("id"))
    .build();
//返回一页查询结果
Page<SampleEntity> sampleEntities =
    elasticsearchTemplate.queryForPage(searchQuery,SampleEntity.
    class);
```

可以使用实体类定义索引结构，代码如下：

```
import org.springframework.data.annotation.Id;
import org.springframework.data.annotation.Version;
import org.springframework.data.elasticsearch.annotations.Document;
import org.springframework.data.elasticsearch.annotations.Field;
import org.springframework.data.elasticsearch.annotations.FieldType;

//通过注解映射实体类到索引库
@Document(indexName = "book",type = "book" , shards = 1, replicas = 0,
indexStoreType = "memory", refreshInterval = "-1")
public class Book {

    @Id
    private String id;                              //图书编号
    private String name;
    private Long price;
    @Version
    private Long version;                           //版本号

public Map<Integer, Collection<String>> getBuckets() {
    return buckets;
}

public void setBuckets(Map<Integer, Collection<String>> buckets) {
    this.buckets = buckets;
}
```

```
    //书对应的类别
    @Field(type = FieldType.Nested)
    private Map<Integer, Collection<String>> buckets = new HashMap();

        public Book(){}

        public Book(String id, String name,Long version) {
            this.id = id;
            this.name = name;
            this.version = version;
        }

        //实体 Bean 需要的一些方法
        public String getId() {
            return id;
        }

        public void setId(String id) {
            this.id = id;
        }

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        public Long getPrice() {
            return price;
        }

        public void setPrice(Long price) {
            this.price = price;
        }

        public long getVersion() {
            return version;
        }

        public void setVersion(long version) {
            this.version = version;
        }
    }
```

实际执行索引结构定义：

```
elasticsearchTemplate.putMapping(Book.class);
```

另一个使用注解的例子，代码如下：

```
@Entity(name="content")
@Document(indexName = "lietuim", type = "content")          //索引类型
public class ContentEntity {
    private int id;
```

```
@Field(type = FieldType.text,searchAnalyzer = "standard",
       analyzer = "standard",store = true) //通过注解指定内容所用的分析器
    private String content1;
    @Field(type = FieldType.Double,store = true)    //指定要存储的浮点数
    private Double num1;
    @Field(type = FieldType.Date,store = true)      //日期类型的发布时间
    private Timestamp nowtime;
    @Field(type = FieldType.text,store = false)     //唯一编号
    private String uuid;
    @Field(type = FieldType.Integer,store = false) //受欢迎程度
    private Integer likenum;
    private int cid;

public ContentEntity(int id, String content1, Double num1,
     Timestamp nowtime, String uuid, Integer likenum, int cid){  //构造方法
       this.id = id;
       this.content1 = content1;
       this.num1 = num1;
       this.nowtime = nowtime;
       this.uuid = uuid;
       this.likenum = likenum;
       this.cid = cid;
    }

    public ContentEntity() {
    }

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "id", unique = true, nullable = false)  //生成的唯一编号
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    @Basic
    @Column(name = "content1")
    public String getContent1() {
        return content1;
    }

    public void setContent1(String content1) {
        this.content1 = content1;
    }

    @Basic
    @Column(name = "num1")
    public Double getNum1() {
        return num1;
    }
```

```
    public void setNum1(Double num1) {
        this.num1 = num1;
    }

    @Basic
    @Column(name = "nowtime")
    public Timestamp getNowtime() {
        return nowtime;
    }

    public void setNowtime(Timestamp nowtime) {
        this.nowtime = nowtime;
    }

    @Basic
    @Column(name = "uuid")
    public String getUuid() {
        return uuid;
    }

    public void setUuid(String uuid) {
        this.uuid = uuid;
    }

    @Basic
    @Column(name = "likenum")                    //评分
    public Integer getLikenum() {
        return likenum;
    }

    public void setLikenum(Integer likenum) {
        this.likenum = likenum;
    }

    @Basic
    @Column(name = "cid")                        //类别
    public int getCid() {
        return cid;
    }

    public void setCid(int cid) {
        this.cid = cid;
    }
}
```

可以通过 XML 配置设置存储库。

使用节点客户端连接到服务器端：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:elasticsearch="http://www.springframework.org/schema/data/
      elasticsearch"
      xsi:schemaLocation="http://www.springframework.org/schema/data/
      elasticsearch
```

```
http://www.springframework.org/schema/data/elasticsearch/spring-
elasticsearch.xsd
    http://www.springframework.org/schema/beans http://www.springframework.
    org/schema/beans/spring-beans.xsd">

    <elasticsearch:node-client id="client" local="true"/>

    <bean name="elasticsearchTemplate" class="org.springframework.data.
    elasticsearch.core.ElasticsearchTemplate">
        <constructor-arg name="client" ref="client"/>
    </bean>

</beans>
```

使用 Transport 客户端连接到服务器端：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:elasticsearch="http://www.springframework.org/schema/data/
       elasticsearch"
       xsi:schemaLocation="http://www.springframework.org/schema/data/
       elasticsearch
http://www.springframework.org/schema/data/elasticsearch/spring-elastic
search.xsd
       http://www.springframework.org/schema/beans http://www.springframework.
org/schema/beans/spring-beans.xsd">

    <elasticsearch:repositories base-package="com.xyz.acme"/>
<!--设定连接参数-->
    <elasticsearch:transport-client id="client" cluster-nodes="ip:9300,
    ip:9300" cluster-name="elasticsearch" />

<bean name="elasticsearchTemplate"
      class="org.springframework.data.elasticsearch.core.
      ElasticsearchTemplate">
        <constructor-arg name="client" ref="client"/>
    </bean>

</beans>
```

6.5　REST 基本概念

REST 即表述性状态传递（Representational State Transfer），是 2000 年 Roy Fielding 博士在他的博士论文中提出的一种软件架构风格。REST 是一种针对网络应用的设计和开发方式，可以降低开发的复杂性，提高系统的可伸缩性。

目前在三种主流的 Web 服务实现方案中，因为 REST 模式的 Web 服务相比复杂的 SOAP 和 XML-RPC 来讲明显更加简洁，因此越来越多的 Web 服务开始采用 REST 风格进行设计和实现。

要构建 REST API，必须了解以下 4 点内容。

- 控制器：控制器控制 HTTP 请求与应用程序逻辑之间的交互。
- 资源：指定连接到 Elasticsearch 服务器的参数。
- 链接：翻页链接。
- 如何构建这些链接：在 REST 控制器中使用 spring-data-common 项目的 PagedResourcesAssembler 类，它可以在响应中生成下一页/上一页的链接。

首先，@Configuration 注解是基于 Java 的 Spring 配置使用的主要构件，它本身是使用@Component 进行元注解的，它使得被注解类标注的 Bean 也成为组件扫描的候选项。@Configuration 类的主要目的是作为 Spring IoC 容器的 Bean 定义源。

注解成配置的类不应该是 final 修饰的类，这个类应该有一个没有参数的构造函数。代码如下：

```
@Configuration
@ComponentScan( basePackages = "org.rest" )
@PropertySource({
  "classpath:rest.properties",
  "classpath:web.properties"
})
public class AppConfig{

   @Bean
   public static PropertySourcesPlaceholderConfigurer
     properties() {

      return new PropertySourcesPlaceholderConfigurer();
   }
}
```

使用 HttpMessageConverter 和注解创建 RESTful 服务。代码如下：

```
@Configuration
@EnableWebMvc
public class WebConfig{
   //
}
```

@EnableWebMvc 注解会在类路径中检测到 Jackson 和 JAXB 2 的存在，并自动创建和注册默认的 JSON 与 XML 转换器。

测试 Spring 上下文：

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration(
  classes = { ApplicationConfig.class, PersistenceConfig.class },
  loader = AnnotationConfigContextLoader.class )
public class SpringTest {

   @Test
   public void whenSpringContextIsInstantiated_thenNoExceptions(){
      // When
   }
}
```

Java 配置类仅用@ContextConfiguration 注解指定，新的 AnnotationConfigContextLoader 类从@Configuration 类加载 Bean 中定义。

请注意，这个测试并没有包含 WebConfig 配置类，因为它需要在 Servlet 上下文中运行，但这里并未提供。

@Controller 是 RESTful API 整个 Web 层中的中心构件。示例代码如下：

```
@Controller
@RequestMapping("/foos")
class FooController {

   @Autowired
   private IFooService service;

   @RequestMapping(method = RequestMethod.GET)
   @ResponseBody
   public List<Foo> findAll() {
      return service.findAll();
   }

   @RequestMapping(value = "/{id}", method = RequestMethod.GET)
   @ResponseBody
   public Foo findOne(@PathVariable("id") Long id) {
      return RestPreconditions.checkFound( service.findOne( id ));
   }

   @RequestMapping(method = RequestMethod.POST)
   @ResponseStatus(HttpStatus.CREATED)
   @ResponseBody
   public Long create(@RequestBody Foo resource) {
      Preconditions.checkNotNull(resource);
      return service.create(resource);
   }

   @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
   @ResponseStatus(HttpStatus.OK)
   public void update(@PathVariable( "id" ) Long id, @RequestBody Foo
   resource) {
      Preconditions.checkNotNull(resource);
      RestPreconditions.checkNotNull(service.getById( resource.getId()));
      service.update(resource);
   }

   @RequestMapping(value = "/{id}", method = RequestMethod.DELETE)
   @ResponseStatus(HttpStatus.OK)
   public void delete(@PathVariable("id") Long id) {
      service.deleteById(id);
   }

}
```

控制器的实现类是非公开的，因为它不需要公开。

通常，控制器是依赖关系链中的最后一个——它接收来自 Spring 前端控制器

（DispathcerServlet）的 HTTP 请求，并将这些请求转发到服务层。如果没有使用场景通过直接引用来注入或操纵控制器，那么就没有必要将它声明为公开的类。

请求映射的实际值及 HTTP 方法都用于确定请求的目标方法。@RequestBody 会将方法的参数绑定到 HTTP 请求的正文中，而@ResponseBody 对响应和返回类型执行相同的操作。

来自 Spring Boot 的@RestController 注释基本上是一个快捷方式，可以帮助我们避免必须定义@ResponseBody。代码如下：

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import javax.annotation.Resource;

@Controller
@RequestMapping(value = "/cah")
public class CahController {

    @Resource(name = "cahService")
    private CahService cahService;

    /**
     * 新增潜导
     *
     * @return
     * @throws Exception
     */
    @RequestMapping(value = "/save")
    @ResponseBody
    public Object save() throws Exception {
        JsonResult jsonResult = new JsonResult();

        jsonResult.setCode("00");
        jsonResult.setMessage("msg");

        return jsonResult;
    }
}
```

调用 CahController 的 save 方法：

```
http://localhost:8080/cah/save
```

当对资源进行部分更新时，可以使用 HTTP PATCH。首先定义实现 REST API 以更新具有多个字段的 HeavyResource。代码如下：

```
public class HeavyResource {
    private Integer id;
    private String name;
    private String address;
    ...
```

然后，需要创建使用 PUT 来处理资源的完整更新的端点：

```
@PutMapping("/heavyresource/{id}")
```

```
public ResponseEntity<?> saveResource(@RequestBody HeavyResource
                                                    heavyResource,
            @PathVariable("id") String id) {
    heavyResourceRepository.save(heavyResource, id);
    return ResponseEntity.ok("resource saved");
}
```

这是更新资源的标准端点。

通常我们会由客户端更新地址字段。在这种情况下，我们不想将所有的字段都传给整个 HeavyResource 对象，但是希望通过 PATCH 方法只更新地址字段，因此可以创建一个 HeavyResourceAddressOnly DTO 来表示地址字段的部分更新。

接下来，可以利用 PATCH 方法发送部分更新。代码如下：

```
@PatchMapping("/heavyresource/{id}")
public ResponseEntity<?> partialUpdateName(
       @RequestBody HeavyResourceAddressOnly partialUpdate,
       @PathVariable("id") String id) {

    heavyResourceRepository.save(partialUpdate, id);
    return ResponseEntity.ok("resource address updated");
}
```

使用这种更细粒度的 DTO，可以只发送需要更新的字段，而无须发送整个 Heavy Resource。如果我们有大量这样的部分更新操作，也可以跳过为每个外部创建一个定制的 DTO，而仅使用一个 Map，代码如下：

```
@RequestMapping(value = "/heavyresource/{id}", method = RequestMethod.
PATCH, consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> partialUpdateGeneric(
  @RequestBody Map<String, Object> updates,
  @PathVariable("id") String id) {

    heavyResourceRepository.save(updates, id);
    return ResponseEntity.ok("resource updated");
}
```

该解决方案将为我们提供更多的灵活性来实现 API。

最后为这两个 HTTP 方法编写测试代码。首先，我们要通过 PUT 方法测试完整资源的更新。代码如下：

```
mockMvc.perform(put("/heavyresource/1")
  .contentType(MediaType.APPLICATION_JSON_VALUE)
  .content(objectMapper.writeValueAsString(
    new HeavyResource(1, "Tom", "Jackson", 12, "heaven street")))
  ).andExpect(status().isOk());
```

通过使用 PATCH 方法来实现部分更新。代码如下：

```
mockMvc.perform(patch("/heavyrecource/1")
  .contentType(MediaType.APPLICATION_JSON_VALUE)
  .content(objectMapper.writeValueAsString(
    new HeavyResourceAddressOnly(1, "5th avenue")))
  ).andExpect(status().isOk());
```

还可以为更通用的方法编写一个测试：

```
HashMap<String, Object> updates = new HashMap<>();
updates.put("address", "5th avenue");

mockMvc.perform(patch("/heavyresource/1")
    .contentType(MediaType.APPLICATION_JSON_VALUE)
    .content(objectMapper.writeValueAsString(updates))
  ).andExpect(status().isOk());
```

6.6 使用 Vue.js 开发搜索界面

Vue.js 是一个构建数据驱动的 Web 界面的渐进式框架。为了实现微服务架构，可以结合 Spring 框架封装搜索请求，展现搜索结果。

首先在服务器端安装 Vue 模块，先查看 npm 版本号。

```
# npm -v
```

npm 版本需要 3.0 以上的，如果低于此版本就需要升级。

可以使用 NVM（Node Version Manager）安装或者升级 Nodejs。NVM 是一个 BASH 脚本，用于管理多个发布的 Node.js 版本。使用 NVM，可以安装所选择的任何可用的 Node.js 版本；也可以用它卸载 Node.js，并从一个版本切换到另一个版本。

为了安装 NPM，可以用 root 用户身份运行以下命令：

```
# curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.2/
install.sh | bash
```

上面的命令将把 NVM 库克隆到~/.nvm，并将源代码行添加到配置文件中(~/.bash_profile、~/.zshrc、~/.profile 或者 ~/.bashrc)。

运行.bash_profile：

```
# source ~/.bash_profile
```

验证 NVM 是否已经安装：

```
# command -v nvm
```

如果安装成功，应该输出 nvm。

现在，可以安装 Node.js 和 npm 了。首先，运行以下命令查看可用的 Node.js 版本列表：

```
# nvm ls-remote
```

然后使用 NVM 安装 Node.js。

```
# nvm install node
# nvm list
```

先生成 package.json 文件：

```
# npm init -f
```

然后安装 Vue 模块：

```
# npm install vue -save
```

在全局安装 vue-cli：

```
# npm install -g vue-cli
```

检查 Vue 是否安装成功：

```
# vue -V
```

然后生成一个叫做 Vue-Project 的项目。

```
# vue init webpack Vue-project
```

进入该项目目录：

```
# cd Vue-Project
```

然后启动项目：

```
# npm run dev
```

Unpkg.com 提供基于 npm 的 CDN 链接。https://unpkg.com/vuex 将始终指向 npm 的最新版本。

```
<head>
    <script
src="https://cdn.bootcss.com/vue/2.5.13/vue.min.js"></script>
</head>
<body>
    <div id="app">
      {{ message }}
    </div>
    <!-- JavaScript 代码需要放在尾部（指定的 HTML 元素之后） -->
    <script>
    new Vue({
        el:'#app',
        data: {
            message:'Hello you!'                    //消息
        }
    });
    </script>
</body>
```

提供一个在页面上已存在的 DOM 元素作为 Vue 实例的挂载目标。可以是 CSS 选择器，也可以是一个 HTMLElement 实例。

在实例挂载之后，元素可以通过 vm.$el 来访问。

在 Vue.js 应用中，使用 Axios 调用 REST API 从服务器获取数据或者发布数据到服务器。通过 Axios 请求 Spring Boot 提供的数据：

```
<head>
    <script src="https://cdn.bootcss.com/vue/2.5.13/vue.min.js"></script>
    <script src="https://cdn.bootcss.com/axios/0.17.1/axios.js"></script>
</head>
<body>
<div id="app">
```

```
    {{ message }}
</div>
<!-- JavaScript 代码需要放在尾部（指定的 HTML 元素之后） -->
<script>
    var vm = new Vue({
       el: '#app',
       data: {
          message: 'Hello you!'
       }
    });

    axios({
       method: 'get',
       url: '/',
    }).then(function (response) {
       console.log(response);
       vm.message = response.data;
    }).catch(function (error) {
          console.log(error)
       });

</script>
</body>
```

因为 Axios 使用了 Promise 对象，所以可以将它与异步/等待结合起来，得到一个简洁且易于使用的 API。

elasticsearch.js 是一个可以在 Node.js 和浏览器中运行的 Elasticsearch 客户端。创建一个客户端实例，代码如下：

```
var es = require('elasticsearch');
                              //使用 Node.js 中的 require() 加载 Elasticsearch 模块

var client = new es.Client({
  host: 'localhost:9200'    //访问 Elasticsearch 服务器的连接参数
  ,log: 'trace'
});
```

在指定索引名称和类型上执行搜索功能的函数，代码如下：

```
function search (myIndex, myType, searchText)
  return client.search({
    index: myIndex,
    type: myType,
    body: {
      fields: {},
      query: {
        match: {
          file_content: searchText        //在 file_content 列执行搜索
        }
      }
    }
  }).then(function (resp) {
      return hits = resp.hits.hits;
  }, function (err) {
```

```
    console.trace(err.message);
});

export { search }
```

上面的代码中定义了一个名为 search()的函数并将其导出。请注意，这里还包含返回语句实际返回该函数的 Promise 对象和结果。

然后在 main.js 中，可以通过 search 这个名字导入 search，并使用该函数：

```
// ./main.js
var Vue = require('vue');

Vue.use(require('vue-resource'));

import { search } from './elasticsearch.js';  //导入 search 模块

new Vue({

    el: 'body',

    methods: {
        search: function() {
            var result =
    search('someindex', 'sometype', 'Search text here' ).then(function(res) {
  // 展示搜索结果
  })
        }
    }
});
```

6.7　使用 Vue.js Paginator 插件实现翻页

前端通过表格显示分页的 JavaScript 插件有 jqGrid 和 jQuery EasyUI 等。但是对于搜索结果页来说，表格展示不够灵活。Vue.js Paginator 是一个简单但功能强大的插件，因为它使用户可以控制如何呈现数据，而不是只能使用预定义的表格。Web 服务器可以将生成页面发给浏览器，然后 Vue 用浏览器进行渲染。

可以使用 Vue 和 Java 做一个搜索网站，当浏览器第一次访问该网站时，Web 服务器把静态的 HTML 页面和 JS 文件等发给浏览器，浏览器单击跳转时前端模拟路由，然后使用 JavaScript 的 fetch()方法或者 AJAX 发送 HTTP 请求数据，Java 接收 HTTP 请求后将返回数据给前端的 Vue，Vue 接收请求获取数据，重新渲染显示页面。后端的 Java 服务只负责使用 REST 收发 JSON 数据。

在 http://hootlex.github.io/vuejs-paginator/中可以看到一个具体的例子。在页面中显示以下的动物信息：

```
[
  {
```

```
    "id": 11,                                    //动物编号
    "name": "Macaw"                              //动物的名字
  },
  {
    "id": 12,
    "name": "Parrot"
  },
  {
    "id": 13,
    "name": "Ostrich"
  },
  {
    "id": 14,
    "name": "Pelican"
  },
  {
    "id": 15,
    "name": "Pigeon"
  }
]
```

Vue.js Paginator 插件需要一个 resource_url 生成简单的分页按钮。当每次页面被更改或获取时，都会发出一个事件来更新资源。

例如，展示第 2 页时，https://hootlex.github.io/vuejs-paginator/samples/animals2.json 的内容如下：

```
{
  "current_page": 2,                             //当前页的编号
  "last_page": 3,       //最后一页的编号，可以根据返回结果总数得到最后一页的编号
  "next_page_url":"https://hootlex.github.io/vuejs-paginator/samples/
  animals3.json",                                //下一页
  "prev_page_url":"https://hootlex.github.io/vuejs-paginator/samples/
  animals1.json",                                //上一页
  "data": [                                      //数据
    {
      "id": 6,
      "name": "Black bear"
    },
    {
      "id": 7,
      "name": "Goat"
    },
    {
      "id": 8,
      "name": "Cockatoo"
    },
    {
      "id": 9,
      "name": "Duck"
    },
    {
      "id": 10,
      "name": "Frog"
```

```
    }
  ]
}
```

导入 Vue.js Paginator 插件并在组件中进行定义。代码如下：

```
new Vue({
  el: '#app',
  data () {
    return {
      //资源变量
      animals: [],
      //在这里定义分页 API 的网址
      resource_url: 'http://hootlex.github.io/vuejs-paginator/samples/
      animals1.json'
    }
  },
  components: {
    VPaginator: VuePaginator
  },
  methods: {
    updateResource(data){
      this.animals = data
    }
  }
  ...
});
```

在搜索结果的 HTML 页面中使用 Vue.js Paginator 插件显示结果数据：

```
<v-paginator :resource_url="resource_url" @update="updateResource">
</v-paginator>
```

每当页面被更改或获取时，资源变量将包含返回的数据。

```
<ul>
  <li v-for="animal in animals">
    {{ animal.name }}
  </li>
</ul>
```

可以编写一个简单的 Rest 后端来配合前端分页开发测试。

在 org.arpit.java2blog.springboot.bean 中创建一个名为 Country.java 的 Bean。代码如下：

```
package org.arpit.java2blog.bean;

public class Country{                                    //国家类

 int id;
 String countryName;

 public Country(int i, String countryName) {
 super();
 this.id = i;
 this.countryName = countryName;
 }
 public int getId() {
```

```
return id;
}
public void setId(int id) {
this.id = id;
}
public String getCountryName() {
return countryName;
}
public void setCountryName(String countryName) {
this.countryName = countryName;
}

}
```

在包 org.arpit.java2blog.springboot 中创建下面一个名为 CountryController.java 的控制类：

```
package org.arpit.java2blog.springboot;

import java.util.ArrayList;
import java.util.List;

import org.arpit.java2blog.springboot.bean.Country;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class CountryController {

 @RequestMapping(value = "/countries",
     method = RequestMethod.GET,headers="Accept=application/json")
 public List<Country> getCountries()                    //用于翻页测试
 {
  List<Country> listOfCountries = new ArrayList<Country>();
  listOfCountries=createCountryList();
  return listOfCountries;
 }

 //创建国家列表的工具方法
 public List<Country> createCountryList()
 {
  Country indiaCountry=new Country(1, "India");
  Country chinaCountry=new Country(4, "China");
  Country nepalCountry=new Country(3, "Nepal");
  Country bhutanCountry=new Country(2, "Bhutan");

  List<Country> listOfCountries = new ArrayList<Country>();
  listOfCountries.add(indiaCountry);
  listOfCountries.add(chinaCountry);
  listOfCountries.add(nepalCountry);
  listOfCountries.add(bhutanCountry);
  return listOfCountries;
 }
}
```

6.8　实现搜索接口

本节首先介绍判断输入字符串编码的方法，然后介绍基本的布尔逻辑查询、指定范围的查询，以及搜索结果排序等实现方法。

6.8.1　编码识别

搜索引擎的查询关键词是很重要的一个参数，这个参数是一个查询字符串的 URL 编码。一个非 ASCII 字符的 URL 编码由一个“%”符号后面跟着两个十六进制的数字组成。中文搜索需要判断传入的这个字符串的 URL 编码是 GBK 还是 UTF-8 格式。

在符合 J2EE 标准的 Web 服务器（例如 Tomcat）中，调用 request.getQueryString()方法就可以得到原始提交的参数。比如发送：

```
http://localhost/search.do?query=%B0%A1
```

getQueryString()方法得到的字符串是：

```
query=%B0%A1
```

然后调用编码识别方法，用正确的编码来解码。

```
String input = "%E6%B5%B7%E6%8A%A5%E7%BD%91";
String codingName=getEncoding(input);                          //判断编码
System.out.println(URLDecoder.decode(input, codingName)); //用正确的编码来解码
```

主要的开发工作是根据输入字符串判断编码是 GBK 还是 UTF-8。

GB2312 的字符编码范围在%B0%A1 至%F7%FE 之间，如表 6-1 所示。

表 6-1　汉字编码对照表

字　符	编　码
啊	%B0%A1
阿	%B0%A2
鞍	%B0%B0
齄	%F7%FE

汉字 Unicode 编码范围从\u4e00 到\u9fa5。UTF-8 汉字 URL 编码后的取值范围在：

```
%E4%B8%80 - %E4%BF%BF
%E5%B8%80 - %E5%BF%BF
%E6%B8%80 - %E6%BF%BF
%E7%80%80 - %E7%BF%BF
%E8%80%80 - %E8%BF%BF
%E9%80%80 - %E9%BE%A5
```

像左括号和右括号这样的 ASCII 编码小于 128 的字符编码都小于%80，例如左括号字

符编码是%28，右括号字符编码是%29。而所有的汉字编码，无论是 UTF-8 或 GBK，每个字节的编码都是大于或等于%80。

```
//判断是否可能是 UTF-8 编码的汉字
public static boolean isUtf8(String code1,String code2,String code3) {
    if (code1.compareTo("E4") >= 0 && code1.compareTo("E9") <= 0 &&
          code2.compareTo("80") >= 0 && code2.compareTo("BF") <= 0 &&
          code3.compareTo("80")>=0 &&code3.compareTo("BF")<=0) {
        return true;
    }
    return false;
}
//判断是否可能是 GB2312 编码的汉字
public static boolean isGb2312(String code1,String code2) {
    if (code1.compareTo("B0") >= 0 && code1.compareTo("F7") <= 0 &&
          code2.compareTo("A0")>=0 &&code2.compareTo("FF")<=0) {
        return true;
    }
    return false;
}
//根据字符列表猜测字符编码
public static String getEncodeByList(List<String> code) {
if(code.size() >= 2 && code.size()%2 == 1 && code.size()%3 == 0) {
    return "utf8";
}
else if(code.size() >= 2 && code.size()%2 == 0 && code.size()%3 != 0) {
    return "gbk";
}
else if(code.size()%6 == 0) {
    for(int m=0;m<code.size();m = m+6) {
        if( ! isUtf8(code.get(m), code.get(m+1), code.get(m+2)) &&
          isGbk(code.get(m), code.get(m+1)) &&
          isGbk(code.get(m+2), code.get(m+3)) ) {
            return "gbk";
        } else if(isUtf8(code.get(m), code.get(m+1), code.get(m+2)) &&
              ! isGbk(code.get(m), code.get(m+1)) ) {
            return "utf8";
        }
        if(! isUtf8(code.get(m+3), code.get(m+4), code.get(m+5)) &&
          isGbk(code.get(m+2), code.get(m+3)) &&
          isGbk(code.get(m+4), code.get(m+5)) ) {
            return "gbk";
        } else if(isUtf8(code.get(m+3), code.get(m+4), code.get(m+5)) &&
          !isGbk(code.get(m+2), code.get(m+3))) {
            return "utf8";
        }
    }
}
return "utf8";
}
```

根据有限状态机的思想把字符串切分成数组。首先定义状态类：

```
public enum CharType {
    Enter,                                          //碰到%
```

```
    Code1,                                          //碰到%后的第一个字符
    Code2,                                          //碰到%后的第二个字符
}
```

然后根据上一个状态及当前的字符决定下一个状态。进入下一个状态时，有可能执行判断字符编码的动作。代码如下：

```
public static String getURLEncoding(String url) {
List<String> codes = new ArrayList<String>();
CharType currentSate = null;                        //记录当前状态
char c1='\0';
char c2='\0';
for(int i=0; i<url.length(); ++i) {
    char currentChar = url.charAt(i);
    if(currentChar == '%') {
        if(currentSate == CharType.Code2 ) {        //第二个字符
            char[] s1 = {c1,c2};
            codes.add(new String(s1));
        }
        currentSate = CharType.Enter;
    }else if(currentSate==CharType.Enter) {         //进入新的状态
        c1 = currentChar;
        currentSate = CharType.Code1;
    }else if(currentSate==CharType.Code1)  {        //第一个字符
        c2 = currentChar;
        currentSate = CharType.Code2;
    }else if(currentSate==CharType.Code2)  {        //第二个字符
        char[] s1 = {c1,c2};
        codes.add(new String(s1));
        currentSate = null;
        return getEncodeByList(codes);              //返回猜测的字符编码
    }
}
if(currentSate==CharType.Code2) {
    char[] s1 = {c1,c2};
    codes.add(new String(s1));
}
return getEncodeByList(codes);                      //返回猜测的字符编码
}
```

6.8.2　布尔搜索

为了实现字词混合搜索，先根据单字列和词列得到两个查询对象，然后再通过BoolQuery.should()方法整合这两个查询对象。代码如下：

```
static void addShould(BoolQueryBuilder qb, PageContext context,String
argName,
    String singleField,String wordField) {
String qString = context.getRequest().getParameter(argName);
                                                    //得到查询参数
if (StringUtils.isNotEmpty(qString)) {      //首先判断输入查询串是非空的字符串
    MatchPhraseQueryBuilder singleQB =
```

```
            QueryBuilders.matchPhraseQuery(singleField, qString);  //字查询
        MatchPhraseQueryBuilder wordQB =
            QueryBuilders.matchPhraseQuery(wordField, qString);    //词查询

        QueryBuilder currentQB =
             QueryBuilders.boolQuery().should(singleQB).should(wordQB);
        qb.should(currentQB);
    }
    }
```

然后调用 addShould()方法合并多个查询条件。

```
BoolQueryBuilder qb = QueryBuilders.boolQuery();

//从输入参数 repname 得到"repnameS"和"repname"列的字词混合查询对象
addShould(qb, context, "repname","repnameS","repname");
//从输入参数 repeditor 得到" repeditorS "和"repeditor"列的字词混合查询对象
addShould(qb, context, "repeditor","repeditorS","repeditor");
```

6.8.3 搜索结果重定向

为了统计用户访问哪条搜索结果，可以让搜索结果页展示统计的用户访问的网址。由统计的用户访问的网址再重定向到显示文档的网址。

请求重定向到另一个页面的最简单的方法是使用响应对象的 sendRedirect()方法。例如：

```
response.sendRedirect("http://www.lietu.com");
```

sendRedirect()方法将状态码和新页面的响应发送回浏览器。也可以同时使用 setStatus()和 setHeader()方法来实现同样的效果。

```
String site = "http://www.lietu.com";

response.setStatus(HttpServletResponse.SC_MOVED_TEMPORARILY);
response.setHeader("Location", site);
```

下面的例子中展示 Servlet 如何执行页面重定向到另一个网址。代码如下：

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PageRedirect extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse
    response)
            throws ServletException, IOException {

        //设置响应内容类型
        response.setContentType("text/html");

        //将被重定向的新网址
        String site = new String("http://www.lietu.com");
```

```
            response.setStatus(HttpServletResponse.SC_MOVED_TEMPORARILY);
            response.setHeader("Location", site);
        }
    }
```

现在来编译上面 Servlet 并在 web.xml 文件中创建以下条目：

```
...
<servlet>
   <servlet-name>PageRedirect</servlet-name>
   <servlet-class>PageRedirect</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>PageRedirect</servlet-name>
   <url-pattern>/PageRedirect</url-pattern>
</servlet-mapping>
...
```

现在使用 URL 的 http://localhost:8080/PageRedirect 调用 Servlet 接口，对 URL 的访问会使页面重定向到 http://www.photofuntoos.com 页面上。

6.8.4　搜索结果排序

搜索结果可以按单列或者多列排序，但是需要保证排序列是不做切分处理的，也就是对该列做索引的时候设置 Field.Index.NOT_ANALYZED。例如 url 网址列没有做过切分，可以按该列排序，而标题列 title 做过切分，不能按该列排序。

有时经常需要按日期倒序排序，为了支持对日期列排序，需要把日期转换成统一的字符串格式“yyyyMMddHHmmssSSS”。如果日期精度低，字符串长度相应变短。

索引日期的示例如下：

```
Date pubDate = rs.getDate("pubDate");
Field f = new Field("pubDate",
    DateTools.dateToString(pubDate, DateTools.Resolution.DAY),//精度到天
    Field.Store.YES,
    Field.Index.NOT_ANALYZED);
```

按日期倒序排序的示例如下：

```
Sort sort= new Sort(new SortField("pubDate",SortField.STRING,true));
ScoreDoc[] hits = searcher.search(query,null,1000,sort).scoreDocs;
```

也可以对多个字段排序，如先按地区列 area 排序，然后按类别 type 排序：

```
Sort sort= new Sort(new SortField[]{new SortField("area"),
new SortField("type")});
ScoreDoc[] hits = searcher.search(query,null,1000,sort).scoreDocs;
```

也可以通过 SortComparatorSource 自定义排序方法。

6.8.5 实现相似文档搜索

有时候需要检索与给定文档（如 BBS 讨论区内某一帖子）相似的文档。例如，打开一个新闻网页，往往有块区域会显示和这篇新闻相关的新闻。对一个卖商品的网站来说，当顾客正在浏览一件商品时，如果能把和这件商品性能、作用很相近的商品也同时罗列在网页的左边，如果顾客想要的商品正好就在其中，那么这个网站的营业额肯定会有所提高。

例如，找出和指定电影相关的电影，代码如下：

```
{
    "more_like_this" : {
        "fields" : ["title", "description"],                    //查询列
        "docs" : [                                              //输入文档
        {
            "_index" : "imdb",                                  //索引名称
            "_type" : "movies",                                 //索引类型
            "_id" : "1"                                         //输入文档编号
        },
        {
            "_index" : "imdb",
            "_type" : "movies",
            "_id" : "2"
        }],
        "min_term_freq" : 1,                                    //最小词频
        "max_query_terms" : 12                                  //查询词最多 12 个
    }
}
```

实现原理是：构造一个 MoreLikeThis(MLT)的对象 mlt，然后调用 mlt.like(1)，这里的 1 是 Lucene 内部的文档编号。然后搜索一下，取前几个结果就是与此文档最为相似的。

like(int docNum)方法返回的 Query 是怎么产生的呢？它首先根据传入的 docNum 找出该文档里去除停用词后的高频词，然后用这些高频词生成 Query，最后把 Query 传进 search 方法得到最后的结果。主要思想就是认为这些高频词足以表示文档信息，然后通过搜索得到最后与此 doc 类似的结果。

另外一个简单的用法是要求提供与给定的文本类似的文档。代码如下：

```
GET /_search
{
    "query": {
        "more_like_this" : {
            "fields" : ["title", "description"],
            "like" : "Once upon a time",                        //查询文本
            "min_term_freq" : 1,
            "max_query_terms" : 12
        }
    }
}
```

在构造 MLT 之前，将最小词频和最小文档频率应用于输入词选择中。如果输入文本中只有一个 apple，那么就不符合 MLT 的限制，因为最小词频设置为 2，如果将输入更改为 apple apple，就能起作用了。代码如下：

```
POST /test_index/_search
{
  "query": {
    "more_like_this": {
      "fields": [
        "text"
      ],
      "like_text": "apple apple",                    //输入文本
      "min_term_freq": 2,                            //最小词频
      "percent_terms_to_match": 1,                   //最少匹配多少个词
      "min_doc_freq": 1                              //最小文档频率
    }
  }
}
```

Java API 调用 MLT 的代码如下：

```
String[] fields = { "title", "description" };          //查询列
String[] likeTexts = { "Once upon a time" };           //查询文本
Item[] likeItems = {new MoreLikeThisQueryBuilder.Item()};
MoreLikeThisQueryBuilder queryBuilder = new MoreLikeThisQueryBuilder(
    fields, likeTexts, likeItems);                     //查询对象
```

6.9　Suggester 搜索词提示

Suggesters（搜索词提示推荐器）基本的运作原理是将输入的文本分解为词，然后在索引的字典里查找相似的词并返回。根据使用场景的不同，Elasticsearch 里设计了 4 种类别的 Suggester，分别是 Term Suggester、Phrase Suggester、Completion Suggester 和 Context Suggester。

Term Suggester 只基于切分出来的单个词提供建议，并不会考虑多个词之间的关系。API 调用方只需为每个词挑选 options（选项）里的词，然后组合在一起返回给用户前端即可。

Phrase Suggester 在 Term suggester 的基础上会考量多个词之间的关系，比如是否同时出现在索引的原文里，其相邻程度及词频如何等。

Completion Suggester 使用有限状态转换（FST）查找提示词。FST 会被 Elasticsearch 装载到内存中进行前缀查找，速度极快，因此 FST 只能用于前缀查找。

Context Suggester 根据索引中所有的文档来推荐查询词，但通常希望通过某些标准来筛选和（或）提升提示词。例如，想要根据品牌提示过滤商品，或者想根据歌曲的风格提示歌曲名称。

为了实现建议过滤和（或）提升，可以在配置完字段时添加上下文映射。可以为自动完成字段定义多个上下文映射。每个上下文映射都有唯一的名称和类型（有文本类别和地理两种类型）。上下文映射在字段映射的 contexts 参数下进行配置。

下面定义类型，每个类型自动完成字段的一个上下文映射。

```
PUT place
{
    "mappings": {
        "shops" : {                                          //商品索引库
            "properties" : {
                "suggest" : {
                    "type" : "completion",                   //自动完成字段
                    "contexts": [                            //上下文映射
                        {
                            "name": "place_type",
                            "type": "category"
                        }
                    ]
                }
            }
        }
    }
}
```

然后定义名为 place_type 的类别上下文，从 cat 字段读取类别。

```
PUT place_path_category
{
    "mappings": {
        "shops" : {
            "properties" : {
                "suggest" : {
                    "type" : "completion",
                    "contexts": [
                        {
                            "name": "place_type",
                            "type": "category",
                            "path": "cat"                    //从 cat 字段读取类别
                        }
                    ]
                }
            }
        }
    }
}
```

通过类别上下文，可以将一个或多个类别与索引时的提示词相关联。在查询时，可以通过相关类别过滤和提升提示词。

如果定义了路径（path），那么将从文档中的路径中读取类别，否则必须在提示字段中送出。代码如下：

```
PUT place/shops/1
{
```

```
    "suggest": {
        "input": ["timmy's", "starbucks", "dunkin donuts"],   //输入提示词
        "contexts": {
            "place_type": ["cafe", "food"]                    //直接给出提示词列表
        }
    }
}
```

6.9.1　拼音提示

为了支持汉语拼音感应，需要把所有的词生成为拼音列。Trie 树可以看成关键词和值的映射。拼音列和词本身都可以作为关键词，值这一列则存放词原型。例如对于“厦门”这个词，会存储两个关键词和值的映射。

```
xiamen -> 厦门
厦门 -> 厦门
```

这样当用户输入“厦”或“xia”时都可能提示出“厦门”这个词。

对于基本的中文词提示来说，关键词和值都是一样的。另外，注音程序把中文词转换成拼音，这部分数据支持汉语拼音感应功能。

因为存在多音字，按词注音会有好的结果。可以在 Trie 树的值域中存储一个词对应的拼音。

```
public static String yin(String sentence){ //传入一个字符串作为要处理的对象
int senLen = sentence.length();             //首先计算出传入的这句话的字符长度
int i = 0;                                  //用来控制匹配的起始位置的变量
StringBuilder result = new StringBuilder(senLen);
TernarySearchTrie.MatchRet matchRet = new TernarySearchTrie. MatchRet
("",0);
while (i < senLen){                         //如果 i 小于此句话的长度就进入循环
    boolean match = dic.matchLong(sentence, i, matchRet); //最大长度匹配
    if (match){                             //已经匹配上，按词注音
        i = matchRet.end;
        result.append(matchRet.data);
    } else                                  //如果没有找到匹配上的词，就按单字注音
    {
        result.append(ziYin.zi2Yin(sentence.charAt(i)));
        ++i;                                // 下次匹配点在这个字符之后
    }
}
return result.toString();
}
```

6.9.2　部署总结

提示词词典 suggestDic.txt 可以放在 WEB-INF/classes/dic/路径下。AutoCompleteServlet 程序可以放在 WEB-INF/lib/路径下，通过 web.xml 发布。搜索界面用到的 JavaScript 脚本

jquery.js、jquery.ajaxQueue.js、jquery.autocomplete.css 和 jquery.autocomplete.js 可以放在 ROOT/js 搜索路径下。

根据不同的用户，提示词也不一样。例如，同样输入“大”字，对于影迷，会提示“大话西游”，而对于美食爱好者，可能会提示“大福”（一种日式甜品）。

6.9.3 相关搜索

在相关搜索的方法中，一种是从搜索日志中挖掘字面相似的词作为相关搜索词列表。首先从一个给定的词中挖掘出多个相关搜索词，可以用编辑距离自动机（即程序）从词表中查找一个词的字面相似词，如果候选的相关搜索词很多，就要筛选出最相关的 10 个词。下面利用 Lucene 搜索的方法筛选出与给定词最相关的词。示例如下：

```
private static final String TEXT_FIELD = "text";

/**
 *
 * @param words 候选相关词列表
 * @param word 要找相关搜索词的种子词
 * @return
 * @throws IOException
 * @throws ParseException
 */
static String[] filterRelated(HashSet<String> words, String word) {
StringBuilder sb = new StringBuilder();

for(int i=0;i<word.length();++i){
    sb.append(word.charAt(i));
    sb.append(" ");
}
//建立内存索引
RAMDirectory store = new RAMDirectory();
IndexWriter writer = new IndexWriter(store, new StandardAnalyzer(), true);

    for(String text:words)  {
    Document document = new Document();
    Field textField =
new Field(TEXT_FIELD, text, Field.Store.YES, Field.Index.TOKENIZED);
    document.add(textField);
    writer.addDocument(document);
    }
    writer.close();

    IndexSearcher searcher = new IndexSearcher(store);

QueryParser queryParser = new QueryParser(TEXT_FIELD,
 new StandardAnalyzer());
    Query query = queryParser.parse(sb.toString());//查询给定的词

    Hits hits = searcher.search(query);
```

```
        int maxRet = Math.min(10, hits.length());

        String[] relatedWords = new String[maxRet];
        for (int i = 0; i < maxRet ; i++) {
          Document document = hits.doc(i);
          String text = document.get(TEXT_FIELD);          //查询结果就是相关搜索词
          ystem.out.println(text);
          relatedWords[i]=text;
        }
        searcher.close();
        store.close();

        return relatedWords;
    }
```

整理出以下相关词表，第 1 列是关键词，后续是 10 个以内的相关搜索词：

```
集福轩婚礼%集福轩
手机定位跟踪系统%手机定位系统%手机定位%手机定位仪器
喷绘材料卖店电话%我要喷绘材料卖店电话
厦门房产%厦门租房%厦门新闻%厦门桑拿%房产%青岛房产%厦门%恒雄房产
送水果%送水%水果
三星传真机%三星手机
```

另外一种方法是可以把多个用户共同查询的词看成相关搜索词，需要有记录用户 IP 的搜索日志才能实现，类似推荐系统。例如，超市把尿布与啤酒放在一起卖，因为这是关联规则挖掘出的结果。

对这个结果的解释是：在美国，一些年轻的父亲下班后经常到超市去买婴儿尿布，而他们中有 30%～40%的人同时会为自己买一些啤酒。产生这一现象的原因是：美国的太太们常叮嘱她们的丈夫下班后为小孩买尿布，而丈夫们在买完尿布后又随手带回了他们喜欢的啤酒。

用户搜索“啤酒”的时候，提示他是否还要找“尿布”，就是使用了关联规则挖掘。ARtool 是一个挖掘关联规则的算法工具集。然后可以通过 RelatedEngine 类查找某个关键词的相关词。代码如下：

```
public static void main(String[] args) throws Exception {
    RelatedEngine re =new RelatedEngine(new File("D:/dic/relatedwords.
    txt"));
    String word = "徐家汇";
    String[] relatedWords = re.getRelated(word);  //得到给定词的相关搜索词
    for(String w : relatedWords) {
        System.out.println(w);
    }
}
```

输出相关搜索词如下：

```
上海徐家汇
徐汇
徐家汇价格是
上房徐家汇路附近有吗
```

最后通过自定义的标签（Tag）RelatedTag 在 JSP 页面显示出相关搜索词。

在标签库描述符中定义 Tag：

```
<tag>
  <name>relatedWords</name>
  <tag-class>com.bitmechanic.listlib.RelatedTag</tag-class>
  <description></description>

  <attribute>
     <name>index</name>
     <required>false</required>
     <rtexprvalue>true</rtexprvalue>
  </attribute>

  <attribute>
     <name>url</name>
     <required>false</required>
     <rtexprvalue>true</rtexprvalue>
  </attribute>

  <attribute>
     <name>query</name>
     <required>true</required>
     <rtexprvalue>true</rtexprvalue>
  </attribute>

</tag>
```

最后在 JSP 页面中引用标签：

```
<list:relatedWords index="D:/search/related" url="Search.jsp" query="
<%=query%>"/>
```

6.9.4 再次查找

有时经常需要从搜索结果中缩小范围再次查找信息。一个实现方法是通过加号（+）连接符连接上次查询词和当前查询词。例如，inputstr 记录了上次查询词，queryString 记录了当前查询词。实现代码如下：

```
if (refind) //如果需要再次查找
queryString = " + ("+queryString+") + ("+inputstr+")";
```

使用这个新的查询词就可以实现再次搜索的功能。

6.9.5 搜索日志

搜索日志是用来分析用户搜索行为和信息需求的重要依据。一般记录如下信息：

- 搜索关键字；
- 用户来源 IP；

- 本次搜索返回结果数量；
- 搜索时间；
- 其他需要记录的应用相关信息。

IP 地址是最容易获取的信息，但其局限性也较为明显：伪 IP、代理、动态 IP、局域网共享同一公网 IP 出口……这些情况都会影响基于 IP 来识别用户的准确性，所以 IP 识别用户的准确性比较低，目前一般不会直接采用 IP 来识别用户。

我们可以通过 Cookie 记录用户 ID。Cookie 是从用户端存放的 Cookie 文件记录中获取的，这个文件里面一般在包含一个 Cookieid 的同时也会记下用户在该网站的 Userid（如果网站需要注册登录并且该用户曾经登录过这个网站且 Cookie 未被删除），所以在记录日志文件中的 Cookie 项时候，可以优先去查询 Cookie 中是否含有用户 ID 类的信息。如果存在则将用户 ID 写入日志的 Cookie 项；如果不存在则查找是否有 Cookieid；如果有 Cookieid 则记录；如没有 Cookieid 则记为”-”。这样日志中的 Cookie 就可以直接作为最有效的用户唯一标识符而被统计了。当然这里需要注意该方法只有网站本身才能够实现，因为用户 ID 作为用户隐私信息只有该网站才知道其在 Cookie 的设置及存放位置，第三方统计工具一般很难获取。

通过以上的方法实现用户身份的唯一标识后，我们可以通过一些途径来采集用户的基础信息、特征信息及行为信息，然后为每位用户建立起详细的简介。具体途径有：

- 用户注册时填写的用户注册信息及基本资料；
- 从网站日志中得到的用户浏览行为数据；
- 从数据库中获取的用户网站业务应用数据；
- 基于用户历史数据的推导和预测；
- 通过直接联系用户或者用户调研的途径获得的用户数据；
- 由第三方服务机构提供的用户数据。

通过用户身份识别及用户基本信息的采集，可以通过网站分析的各种方法在网站中实现一些有价值的应用：

- 基于用户特征信息的用户细分；
- 基于用户的个性化页面设置；
- 基于用户行为数据的关联推荐；
- 基于用户兴趣的定向营销。

为了不影响即时搜索的速度，一般不把搜索日志记录直接记录在数据库中，而是写在文本文件中。这里推荐使用 Logback（http://logback.qos.ch/）的日志功能来实现。Logback 提供了 3 个 jar 包，分别是 Core、classic 和 access。其中 Core 是基础，其他两个包依赖于这个包。logback-classic 是 SLF4J 原生的实现，所以可以用其他 logging 系统去替换它。当然 logback-classic 依赖于 slf4j-api。logback-access 与 servlet 容器集成，提供 http-access 的 log 功能。SLF4J（http://www.slf4j.org/）几乎已经称为业界日志的统一接口。

这里的项目一共需要 3 个包：slf4j-api-1.6.1.jar、logback-classic-0.9.21.jar 和 logback-

core-0.9.21.jar。Logback 通过 logback.xml 进行配置。

这里把当前日志写到 D:/logs/log 文件中，新一天的日志开始的时候，前一天的日志将生成一个新文件。

在搜索类中初始化日志类：

```
private static Logger logger = LoggerFactory.getLogger(SearchBbs.class);
```

然后当用户执行一次搜索时，记录查询词、返回结果数量、用户 IP 及查询时间等：

```
logger.info(_query+"|"+desc.count+"|"+"bbs"+"|"+ip);
```

日志文件 log.txt 记录的结果示例如下：

```
什么是新生儿|37|topic|124.1.0.0|2007-11-21 12:25:36
什么是新生儿|28|bbs|124.1.0.0|2007-11-21 12:25:42
怀孕|18|topic|124.1.0.0|2007-11-21 12:26:05
怀孕|2|shangjia|124.1.0.0|2007-11-21 12:26:05
怀孕|145|bbs|124.1.0.0|2007-11-21 12:26:06
怀孕|18|topic|124.1.0.0|2007-11-21 12:30:33
```

其中，第 1 列是用户搜索词，第 2 列是搜索返回结果数量，第 3 列是搜索类别，第 4 列是 IP 地址，第 5 列是搜索的时间。

然后定义搜索日志统计表，例如我们需要统计搜索最多的词，可以把搜索最多的词放在 keywordAnalysis 表中，如下：

```
CREATE TABLE [keywordAnalysis] (
    [searchTerms] [varchar] (50)  NOT NULL ,    --搜索词
    [AccessCount] [int] NULL ,                  --搜索计数
    [Result] [int] NULL                         --该词返回结果数
)
```

6.10 Word2vec 挖掘相关搜索词

可以把词作为特征，通过 Word2vec 把特征映射到 K 维向量空间，为文本数据寻求更深层次的特征表示。

Word2vec 使用的是 Distributed representation 的词向量表示方式。其基本思想是通过训练将每个词映射成 K 维实数向量（K 是模型中的超参数），通过词之间的距离（如 cosine 相似度、欧氏距离等）来判断它们之间的语义相似度，其采用一个三层的神经网络，即输入层—隐层—输出层。

首先，算法对整个语料库中的每个单词计数。然后将词汇放入排序数组，最常见的词排在最前面。接着构建一个哈夫曼树，将低频词组合成具有内部节点的更长的根到叶子节点的路径，这样做的结果是频率最高的词有最短的编码。

如图 6-1 所示，可以沿着哈夫曼树从根节点一直走到叶子节点的词 w_2 处。

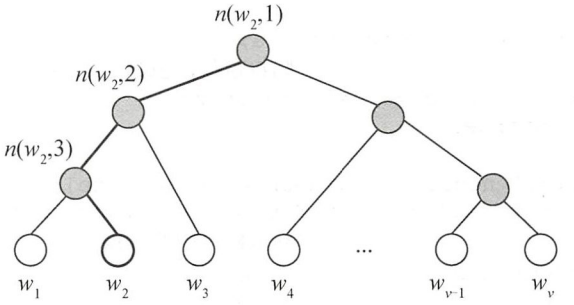


图 6-1　哈夫曼树

在哈夫曼树中，采用二元逻辑回归的方法计算核心词的概率，即规定沿着左边的子树走，那么就是负类（哈夫曼树编码 1），沿着右边的子树走，那么就是正类（哈夫曼树编码 0）。判别正类和负类的方法使用 sigmoid()函数，即：

$$P(+)=\sigma(\boldsymbol{x}_w^{\mathrm{T}}\theta)=\frac{1}{1+\mathrm{e}^{-\boldsymbol{x}_w^{\mathrm{T}}\theta}}$$

其中，$\boldsymbol{x}_w$ 是训练样本$(w, \text{context}(w))$中 context(w)的词向量，而 θ 则是我们需要从训练样本求出的逻辑回归的模型参数。那么被划分为左边的子树而成为负类的概率为 $1-P(+)$，对于图 6-1 中的 w_2，其哈夫曼编码为 110，我们可以得到其概率为：

$(1-P(+))\theta_0*(1-P(+))\theta_1*P(+)\theta_2$

在 w_2 的计算路径中共分 3 步完成，这样进行梯度计算的公式为：

$$\prod_{i=1}^{3}p(w_2)=(1-\frac{1}{1+\mathrm{e}^{\boldsymbol{x}_w^{\mathrm{T}}\theta_0}})(1-\frac{1}{1+\mathrm{e}^{\boldsymbol{x}_w^{\mathrm{T}}\theta_1}})\frac{1}{1+\mathrm{e}^{\boldsymbol{x}_w^{\mathrm{T}}\theta_2}}$$

模型的目的就是最小化上面的概率。针对一个训练样本(w2,context(w2))，上式中 $\boldsymbol{x}_w$ 为 w_2 语境中的词，θ 为内部节点的参数。对于所有的训练样本，我们期望最大化所有样本的似然函数乘积。

skip-gram 模型在 Word2vec 中的具体实现如下。

输入：已经完成分词的中文语料或者英文语料，词向量的维度大小为 200，窗口大小（即多少个词）为 5，Skip-gram 的上下文大小 2c=8，步长 alpha=0.025。

输出：所有的词向量 $\boldsymbol{w}$，哈夫曼树的内部节点模型参数 θ。

实现代码如下：

```
private void skipGram(int index, List<WordNeuron> sentence, int b,
double alpha) {
      WordNeuron word = sentence.get(index);
                              //word 为待处理词，即样本(w,context(w))中的 w
      int a, c = 0;
      for (a = b; a < windowSize * 2 + 1 - b; a++) {
                              //b 为随机赋值，b∈{1,2,3,4}
         if (a == windowSize) {
            continue;
```

```
            }
            c = index - windowSize + a;
                                    //随机赋值，c∈{1,2,3,4}，因此称为 skip-gram
            if (c < 0 || c >= sentence.size()) {
                continue;
            }
            double[] neu1e = new double[vectorSize];          //误差项
            List<HuffmanNode> pathNeurons = word.getPathNeurons();
                                    //获取路径中的节点
            WordNeuron we = sentence.get(c);                  //获取窗口词
            for (int neuronIndex = 0; neuronIndex < pathNeurons.size() - 1;
            neuronIndex++){
 //获取待处理词的内部节点 θ
HuffmanNeuron out = (HuffmanNeuron) pathNeurons.get(neuronIndex);
                double f = 0;
                for (int j = 0; j < vectorSize; j++) {
                    f += we.vector[j] * out.vector[j];
                }
                if (f <= -MAX_EXP || f >= MAX_EXP) {
                    continue;
                } else {
                    f = (f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2);
                    f = expTable[(int) f];
                }
              HuffmanNeuron outNext = (HuffmanNeuron) pathNeurons.get
              (neuronIndex+1);
                double g = (1 - outNext.code - f) * alpha;
                                    // g 为梯度和学习率的乘积
                for (c = 0; c < vectorSize; c++) {
                    neu1e[c] += g * out.vector[c];
                }
                 for (c = 0; c < vectorSize; c++) {
                    out.vector[c] += g * we.vector[c];    //更新内部节点参数 θ
                }
             }
            for (int j = 0; j < vectorSize; j++) {
                we.vector[j] += neu1e[j];                 //更新窗口中的词
            }
        }
    }
```

可以使用最大堆（PairingHeap）找距离最近的 k 个词。

首先用爬虫抓取学习用的语料库。语料库格式是：每行一句话，每句话中的词用空格分开。使用 HttpClient 从必应词典抓取句子的代码如下：

```
public static ArrayList<String> getPairsByWordAndOffset(String word,
int pageOffet,
        CloseableHttpClient httpclient) throws Exception {
String urlAjax = "http://cn.bing.com/dict/service?q="
        + URLEncoder.encode(word, "UTF-8") + "&offset="
        + URLEncoder.encode(pageOffet + "", "UTF-8") + "&dtype=sen";
HttpGet httpgetAjax = new HttpGet(urlAjax);               //发送 GET 请求
HttpResponse responseAjax = httpclient.execute(httpgetAjax);
```

```
                                                        //得到响应结果
    HttpEntity entityAjax = responseAjax.getEntity();

    ArrayList<String> ret = new ArrayList<String>();
    if (entityAjax != null) {
        //读入内容流并以字符串形式返回，这里指定网页编码是 UTF-8
        //网页的 Meta 标签中指定了编码
        String content = EntityUtils.toString(entityAjax, "utf-8");
        Document doc = Jsoup.parse(content);
        Elements elementsLi = doc.select(".se_li");
        if (elementsLi == null || elementsLi.size() == 0) {
            return null;
        }
        // Jsoup 处理提取目标内容
        for (Element pairs : elementsLi) {
            if (pairs.select(".se_li1").size() == 0) {
                continue;
            }
            Element s = pairs.select(".se_li1").first();

            if (s.select(".sen_cn").size() == 0) {
                continue;
            }
            Element senCn = s.select(".sen_cn").first();    //找出中文例句

            List<Element> nodes = senCn.children();

            StringBuilder sb = new StringBuilder();         //句子缓存

            for(Element n:nodes){
                sb.append(n.text() +" ");                   //空格分隔词
            }
            ret.add(sb.toString().trim());
        }
        EntityUtils.consume(entityAjax);                    // 关闭内容流
    }

    return ret; //返回所有的句子
    }
```

机器学习软件包 Deeplearning4J（下载地址是 https://deeplearning4j.org/）中包含了 Word2vec 的实现。新建一个项目，使用 Maven 导入 Deeplearning4J 相关的 jar 包。

```
<properties>
   <nd4j.version>0.7.1</nd4j.version>
   <dl4j.version>0.7.1</dl4j.version>
</properties>

<dependencies>
   <dependency>
      <groupId>org.deeplearning4j</groupId>
      <artifactId>deeplearning4j-ui-model</artifactId>
```

```
        <version>${dl4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.deeplearning4j</groupId>
        <artifactId>deeplearning4j-nlp</artifactId>
        <version>${dl4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.nd4j</groupId>
        <artifactId>nd4j-native</artifactId>
        <version>${nd4j.version}</version>
    </dependency>
</dependencies>
```

使用如下的代码挖掘英文相关词。

```
public class Word2VecDemo {

    private String inputFilePath = "input/";
    private String modelFilePath = "output/word2vec.bin";

    public static void main(String[] args) throws IOException {

        Word2VecDemo word2VecDemo = new Word2VecDemo();
        word2VecDemo.train();                              //训练模型
        //读入模型
        Word2Vec word2VecModel =
            WordVectorSerializer.readWord2VecModel(new
File(word2VecDemo.modelFilePath));
        //返回 10 个最相关的词
        Collection<String> list = word2VecModel.wordsNearest("boy" , 10);
        System.out.println(" boy: "+ list);          //输出相关词列表
    }

    public  void train() throws IOException {
       //使用换行作为句子分隔符号
       SentenceIterator sentenceIterator =
             new FileSentenceIterator(new File(inputFilePath));
        TokenizerFactory tokenizerFactory = new DefaultTokenizerFactory();
        tokenizerFactory.setTokenPreProcessor(new CommonPreprocessor());

        Word2Vec vec = new Word2Vec.Builder()
               .minWordFrequency(2)
               .layerSize(300)
               .windowSize(5)                              //设定窗口大小
               .seed(42)
               .epochs(3)
               .elementsLearningAlgorithm(new SkipGram<VocabWord>())
                                                           //学习算法
               .iterate(sentenceIterator)
               .tokenizerFactory(tokenizerFactory)
```

```
                .build();                                    //构建训练器
        vec.fit();
        //保存模型
        WordVectorSerializer.writeWord2VecModel(vec, "output/word2vec.bin");
    }
}
```

6.11　部署网站

服务器端操作系统推荐采用 Linux。Linux 有各种版本，这里以免费的 CentOS 为例。我们用静态 IP 地址配置一个网络连接的 IPv4 属性。例如，静态 IP 地址是 201.147.214.149。eth0 配置文件的内容如下：

```
# cat /etc/sysconfig/network-scripts/ifcfg-eth0
TYPE=Ethernet
DEVICE=eth0
HWADDR=00:2g:fc:1b:c3:9e
ONBOOT=yes
USERCTL=no
IPV6INIT=no
PEERDNS=yes
NETMASK=255.255.255.128
IPADDR=201.147.214.149
GATEWAY=201.147.214.254
```

重新启动网络服务：

```
#service network restart
```

如果托管的机器要更换，可以先远程设置好网卡的配置，然后再更换机器，重新启动机器，同时修改 DNS 中的 IP 地址。

6.11.1　部署到 Web 服务器

配置 Java 环境，设置环境变量 JAVA_HOME 和 PATH 的值。修改脚本文件/etc/bashrc：

```
#vi /etc/bashrc
```

增加如下行：

```
export JAVA_HOME=/usr/local/jdk1.8.0_21
export PATH=$JAVA_HOME/bin:$PATH
```

从 Tomcat 官方网站 http://tomcat.apache.org/下载 tar.gz 包：

```
# wget
http://www.fayea.com/apache-mirror/tomcat/tomcat-8/v8.0.33/bin/apache-
tomcat-8.0.33.tar.gz
```

然后解压缩这个文件：

```
#tar -xf apache-tomcat-8.0.33.tar.gz
```

然后增加 Tomcat 所使用的内存。修改配置文件 catalina.sh 如下：

```
#vi /usr/local/apache-tomcat-8.0.33/bin/catalina.sh
```

在文件 catalina.sh 的开始位置增加如下行：

```
JAVA_OPTS=-Xmx1024m
```

修改 Tomcat 配置文件 server.xml，把监听端口号从 8080 改到 80，并且支持 UTF-8 编码：

```
#vi /usr/local/apache-tomcat-8.0.33/conf/server.xml
```

增加配置：

```
useBodyEncodingForURI="true" URIEncoding="UTF-8"
```

可以把 Web 应用打一个 war 包，然后传到服务器上的 webapps/子路径下，程序会自动解压缩 war 包中的 Web 应用。也可以压缩开发环境中的文件：

```
#tar -cjf price.tar.bz2 ./price
```

在正式环境中下载压缩好的文件 price.tar.bz2，然后解压缩文件：

```
#tar -xjf price.tar.bz2
```

再启动 Tomcat：

```
#startup.sh
```

查看 Tomcat 是否已经启动了：

```
#pgrep -l java
```

或者使用如下命令：

```
#ps -ef |grep java
```

虚拟主机服务器提供商可能提供这样的服务：当 Tomcat 服务停止的时候，会自动启动一个 Apache 显示错误信息。启动 Tomcat 之前，先停止 Apache 服务：

```
#httpd -k stop
```

查看 Apache 服务是否已经停止了：

```
#pgrep httpd
```

如果需要更好的性能，可以使用 Resin。处理静态页面，Resin 比 Ngnix 或者 Apache 快。可以在 http://www.caucho.com/download/上下载 Resin 免费版本。

在 bin 目录下，用 vi 命令新建一个名为 startResin.sh 的文件：

```
#vi ./startResin.sh
```

在文件内输入如下信息：

```
export LC_ALL=zh_CN.GB18030
export LANG=zh_CN.GB18030
nohup ./httpd.sh & -Xms512M -Xmx1024M
```

然后备份到目录/home/webbak/ROOT2012 下：

```
cp -r ./ROOT/  /home/webbak/ROOT2012
启动 Resin 4。
resin.sh start
```

如果需要，可以给网站买一个好记的域名。向域名供应商购买域名后，增加 DNS 信息。如果修改 DNS 信息后，要清空本地的 DNS 缓存信息。Windows 下可以使用如下的命令清空缓存：

```
>ipconfig /flushdns
```

查询一个域名的 A 记录：

```
>nslookup www.lietu.com 198.153.192.1
```

6.11.2　防止攻击

如果站点无法访问了，可能就是被攻击了。常见的一种攻击方式叫做分布式拒绝服务攻击，即 Distributed denial of service（简称 DDOS）。检查服务器是否在 DDOS 状态的一个快速而有用的命令是：

```
#netstat -anp |grep 'tcp\|udp' | awk '{print $5}' | cut -d: -f1 | sort |
uniq -c | sort -n
```

这会列出与服务器建立连接最多的 IP。但是要记住，DDOS 正在变得更复杂，它可能用更多的 IP 地址，每个 IP 地址使用更少的连接，如使用代理 IP。如果是这样，即使在 DDOS 下，仍然只有很少数量的连接，这就叫做 CC 攻击。

另外一个非常重要的事情是看有多少正在处理的活跃的连接。通过命令：

```
#netstat -n | grep :80 |wc -l
```

将显示活跃的连接数。许多攻击通常是开启一个到服务器的连接，然后不发送任何答复，让服务器等待到超时。活跃连接的数量会有很大的不同，但如果是 500 个以上，就可能有问题。例如：

```
#netstat -n | grep :80 | grep SYN |wc -l
```

如果超过 100 个，就有 SYN 攻击的麻烦。大量的 SYN 请求会导致未连接队列被塞满，使正常的 TCP 连接无法顺利完成三次握手，通过增大未连接队列空间可以缓解这种压力。

Linux 用变量 tcp_max_syn_backlog 定义 backlog 队列容纳的最大半连接数。在 Redhat AS 中，这个值默认是 1024。但这个值是远远不够的，一次强度不大的 SYN 攻击就能使半连接队列占满。可以通过以下命令修改此变量的值：

```
# sysctl -w net.ipv4.tcp_max_syn_backlog="2048"
```

在 linux 下可以通过修改/etc/sysctl.conf，添加下列选项达到效果。

```
# add by geminis for syn crack
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_max_syn_backlog=2048
net.ipv4.tcp_synack_retries=1
```

Linux 有一个很好的工具拒绝为“不怀好意”的 IP 提供服务，叫做 iptables。很多管理员害怕使用 iptables。阻塞 IP 会阻塞这个 IP 访问任何服务器资源，不仅仅是 Web 服务器，还包括 FTP 和 Telnet 等。如果直接编辑 iptables 的配置文件，可能会导致宕机。例如，一个语法错误，可能会阻止用户访问 SSH、FTP、HTTP 和任何其他的服务。因此，不要直接编辑配置文件 iptables-config，最好从 Linux 命令行进入 iptables 命令。如果有语法错误，命令行接口会拒绝这个命令。下面是一些常用的例子：

阻塞从 120.60.0.0 到 120.60.255.255 范围内的 IP。

```
#iptables -I INPUT -m iprange --src-range 120.60.0.0-120.60.255.255 -j DROP
```

要阻止一个 IP，如 120.60.43.201，使用下面的命令：

```
#iptables -A INPUT -s 120.60.43.201 -j DROP
```

显示当前的 iptables 文件而不编辑它，使用下面的命令：

```
#iptables -L
```

iptables 不能自动屏蔽恶意 IP，只能手动屏蔽。一个轻量级的脚本 DDOS deflate 能够自动屏蔽 DDOS 攻击者的 IP。

通过配置/usr/local/ddos/ignore.ip.list，可以配置白名单的 IP 地址。

IP 地址被封时间是预先设定的，默认 600 秒后自动解除封锁。通过配置文件，脚本可以定时周期性运行（默认是 1 分钟）。有 IP 地址被封锁时，可以为指定的邮箱接收电子邮件警报。这些都可以写在配置文件/usr/local/ddos/ddos.conf 中。安装 DDOS deflate：

```
# wget http://www.inetbase.com/scripts/ddos/install.sh
# chmod 0700 install.sh
# ./install.sh
```

下面解释一下 DDOS deflate 脚本主要配置文件 ddos.conf。

```
##### Paths of the script and other files
PROGDIR="/usr/local/ddos"                              //文件存放目录
PROG="/usr/local/ddos/ddos.sh"                         //主要功能脚本
IGNORE_IP_LIST="/usr/local/ddos/ignore.ip.list"        //白名单地址列表
CRON="/etc/cron.d/ddos.cron"                           //crond 定时任务脚本
APF="/etc/apf/apf"
IPT="/sbin/iptables"

##### frequency in minutes for running the script
##### Caution: Every time this setting is changed, run the script with --cron
#####          option so that the new frequency takes effect
FREQ=1                    //间隔多久检查一次，默认 1 分钟

##### How many connections define a bad IP? Indicate that below.
NO_OF_CONNECTIONS=150   //最大连接数设置，超过这个数字的 IP 就会被屏蔽，默认即可

##### APF_BAN=1 (Make sure your APF version is atleast 0.96)
##### APF_BAN=0 (Uses iptables for banning ips instead of APF)
APF_BAN=0                 //1：使用 APF，0：使用 iptables，推荐使用 iptables
```

```
##### KILL=0 (Bad IPs are'nt banned, good for interactive execution
of script)
##### KILL=1 (Recommended setting)
KILL=1                                                    //是否屏蔽 IP，默认即可

##### An email is sent to the following address when an IP is banned.
##### Blank would suppress sending of mails
EMAIL_TO="root"          //发送电子邮件报警的邮箱地址，换成自己使用的邮箱即可

##### Number of seconds the banned ip should remain in blacklist.
BAN_PERIOD=600           //屏蔽 IP 的时间，根据情况调整
```

最后开启系统 crond 服务即可。执行 uninstall.ddos 卸载脚本：

```
# wget http://www.inetbase.com/scripts/ddos/uninstall.ddos
# chmod 0700 uninstall.ddos
# ./uninstall.ddos
```

更好的方法是安装 CSF（Config security firewall），它可以在极大程度上保护服务器的安全。CSF 是可以免费使用的，基于 Iptables 的防火墙，很容易集成到 CPanel。CPanel 是为网站所有者设计的一套 Web 形式的控制系统，网站所有者甚至可以直接把它当作网站后台 Windows 操作系统。

银行为了防止黑客暴力破解持卡人的密码，采用连续三次输入密码错误，就锁定该账户的保护形式。CSF 防火墙为了防止暴力破解密码，也会自动屏蔽连续登录失败的 IP。

可以通过 CSF 管理网络端口，只开放必要的端口，还可以免疫小流量的 DDOS 和 CC 攻击。

CentOS 下需要先安装 CSF 依赖包：

```
#yum install perl-libwww-perl perl iptables
#wget http://www.configserver.com/free/csf.tgz
#tar zxf csf.tar.gz
```

如果和 Apache 服务器配合使用，则执行：

```
#sh ./csf/install.sh
```

如果直接管理防火墙，则执行：

```
#sh ./csf/install.directadmin.sh
```

按照安装程序的指示装好程序后，可以运行测试程序：

```
#perl /etc/csf/csftest.pl
```

如果没问题就可以启动防火墙了。

```
#csf -s
```

重新启动防火墙：

```
#csf -r
```

刷新规则，或者停止防火墙：

```
#csf -f
```

6.12 使用 Rust 开发搜索界面

本节将介绍使用 Rust 语言构建前端 Web 应用。

首先安装 Cargo（Cargo 是 Ruse 的构建系统和包管理工具）：

```
# yum install cargo
```

安装 Cargo Web：

```
# cargo install cargo-web
```

这个 Cargo 子命令旨在使构建、开发和部署在 Rust 中编写的客户端 Web 应用程序更加简单方便。

用以下子命令建立项目：

```
# cargo web build
```

自动在 Web 浏览器中运行测试，命令如下：

```
# cargo web test
```

构建项目，启动嵌入式 Web 服务器并根据需要进行重建，命令如下：

```
# cargo web start
```

如有需要，可以在 Linux 系统下自动下载并安装 Emscripten。

6.13 本章小结

本章介绍了使用 Java Spring 框架实现搜索界面，并且介绍了很多重要的搜索功能界面的实现，如复杂条件搜索界面、用户输入提示词和分类查找界面等。

Searchkit 采用 React 构建。React 起源于 Facebook 公司的内部项目。React 虚拟 DOM 到真实 DOM 的渲染是通过 react.render 全局 API 实现的。

2010 年以前一般使用 Script.aculo.us 框架实现自动完成功能。当 jQuery 开始流行后，依赖 Prototype 的 Script.aculo.us 框架不再流行。jQuery 采用操作 DOM 树的方式实现动态页面展示。

目前，很多搜索界面前端采用 JavaScript 进行开发，等 WebAssembly 技术逐渐成型后，前端界面开发会更加简单。

Pebble 是和 PHP 模板 Twig 类似的 Java 模板引擎，具体功能的实现可以参考 Thymeleaf 或者 Velocity。

在界面设计上，要想办法节约用户的时间。例如，手机上的锁定状态，需要用户额外输入才能解锁，可以用触感指纹或者手势设计来代替。

有些热词会直接跳转。例如，搜索手机，可直接跳转到相关手机的购买界面，而不执行相关性查询，是因为有个散列表对应查询词和跳转的页面。

Spring 框架的第一个版本是由 Rod Johnson 撰写的。他在 2000 年为伦敦的金融界提供独立咨询业务时编写了 Spring 框架最开始的部分。2002 年 10 月 Wrox 公司出版了他的《Java 企业应用设计与开发专家一对一》一书，发布了该框架。该书发表后，基于读者的要求，源代码在开源使用协议下对外提供。由此一批自愿拓展 Spring 框架的程序开发员组成了开发团队，并于 2003 年 2 月在 Sourceforge 上构建了一个项目。在 Spring 框架上工作了一年之后，这个团队在 2004 年 3 月发布了 Spring 框架第一个版本（即 Spring 1.0）。继这个版本之后，Spring 框架在 Java 社区里变得非常流行。

在 Spring Boot 之前的 Spring Web 应用需要打包成 WAR 格式，然后才能部署到 Tomcat 中。Spring Boot 可以使用内嵌的 Tomcat，由 main()方法来启动 Spring，接着启动 Tomcat，而不是由 Tomcat 来启动 Spring。EmbeddedServletContainerAutoConfiguration 类会进行 Tomcat 的配置，由 TomcatEmbeddedServletContainer 类进行启动。

第 7 章　Elastic 栈系统监控

我们可以使用 Elastic 栈（Elastic stack）实现对应用程序日志的传输、处理、管理和搜索。Elastic 栈涉及以下几个组件。

- Beats：用于轻量级日志采集，支持文件采集、系统数据采集和特定中间件数据采集等。
- Logstash：用于日志结构化和标签化，支持用 DSL 方式将数据进行结构化。
- Elasticsearch：用于提供日志的相关索引，使得日志能够有效地被检索。
- Kibana：提供用于日志检索和特定度量展示的面板。
- X-Pack：用于监控与预警相关组件，可以集成到 Elasticsearch 中。
- Curator：用于管理 Elasticsearch 集群索引的相关数据，对索引进行分析。

除了 Elasticsearch，也可以使用 Graylog 管理和搜索日志。本章首先介绍使用 Elastic 栈管理日志，然后介绍 Graylog 日志管理平台。

7.1　管理 Elasticsearch 集群

启动一个 Elasticsearch 实例的时候，就是启动一个节点。连接在一起的节点集合称为一个集群。如果正在运行 Elasticsearch 的单个节点，那么就是一个节点组成的集群。

默认情况下，集群中的每个节点都可以处理 HTTP 和 Transport 流量。Transport 层专用于节点和 Java TransportClient 之间的通信，而 HTTP 层仅由外部 REST 客户端使用。

每个节点都知道集群中的所有其他节点，并能够将客户端请求转发到适当的节点。除此之外，每个节点都有一个或多个目的，见以下说明。

- 符合主节点（Master）资格的节点：一个节点的 node.master 设置为 true（默认值），这使得它有资格被选为控制集群的主节点。
- 数据节点：node.data 设置为 true（默认值）的节点。数据节点保存数据并执行数据相关操作，如 CRUD、搜索和聚合。
- 摄取节点：node.ingest 设置为 true（默认值）的节点。摄取节点能够将摄取流水线应用于文档，以便在索引之前转换和丰富文档。因为摄取负载沉重，所以建议使用专门的摄取节点并将主节点和数据节点标记为 node.ingest:false 是有意义的。

默认情况下，Elasticsearch 集群中的每个节点都有成为主节点的资格，也都存储数据，还可以提供查询服务。这对于小型集群来说非常方便，但是随着集群的发展，考虑将专用

主节点与专用数据节点分开是很重要的。

reroute 命令允许显式地执行包含特定命令的集群重路由分配命令。例如，分片可以从一个节点移动到另一个节点，可以取消分配，或者可以在特定节点上显式地分配未分配的分片。

以下是一个简单的 reroute API 调用的简短示例。

```
POST /_cluster/reroute
{
    "commands" : [
        {
            "move" : {
                "index" : "test", "shard" : 0,
                "from_node" : "node1", "to_node" : "node2"
                                                    //把分片从 node1 转移到 node2
            }
        }

    ]
}
```

如分片丢失，也可以尝试使用 reroute 命令找回丢失的分片。

从 Elasticsearch 5.5 开始，reroute 命令已分成两个不同的命令，即 allocate_replica 和 allocate_empty_primary。

allocate_stale_primary 命令给主分片分配一个包含陈旧副本的节点；allocate_replica 命令给未分配的副本分片分配一个节点；allocate_empty_primary 命令给一个节点分配一个空的主分片。

7.1.1　写入权限控制

如果能让 Elasticsearch 提供只读而不可以写入的端口号，就能够实现权限控制。有一个方法就是把 Nginx 放在 Elasticsearch 前端。Nginx 是一款开源的高性能 HTTP 服务器。

只允许 GET 请求的一个配置例子如下：

```
worker_processes  1;                                    //工作进程数目
pid               nginx.pid;                            //进程标识符存放路径

events {
    worker_connections  1024;                           //工作进程的最大连接数量
}

http {

  server {

    listen       8080;                                  //监听端口
    server_name  search.example.com;                    //访问域名
```

```
    error_log   elasticsearch-errors.log;          //错误日志存放路径
    access_log  elasticsearch.log;                 //访问日志存放路径

    location / {
      if ($request_method !~ "GET") {              //拒绝非 GET 请求
        return 403;
        break;
      }

      proxy_pass http://localhost:9200;            //代理转发
      proxy_redirect off;                          //不重定向

      //将代理服务器收到的用户信息传到真实服务器上
      proxy_set_header  X-Real-IP  $remote_addr;
      proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header  Host $http_host;
    }

  }

}
```

使用这个配置：

```
$ nginx -c path/to/this/file
```

然后测试 Nginx：

```
$ curl -i -X GET http://localhost:8080/_search -d '{"query":{"match_
all":{}}}'
HTTP/1.1 200 OK
```

显示 GET 请求可以正常访问。

```
$ curl -i -X POST http://localhost:8080/test/test/1 -d '{"foo":"bar"}'
HTTP/1.1 403 Forbidden
```

显示 POST 请求无法访问。

```
$ curl -i -X DELETE http://localhost:8080/test/
HTTP/1.1 403 Forbidden
```

显示 DELETE 请求也无法访问。

安装 X-Pack 插件后，所有对 Elasticsearch 的访问都增加了安全机制，即需要提供用户名和密码，默认分别为 elastic 和 changeme。使用 sense 插件访问的时候可以输入用户名和密码，如果是使用 CURL 等方式访问，则需要在 HTTP 的 header 中增加 Authentication 参数。

安全的客户端可以参考 https://github.com/elastic/found-shield-example。

7.1.2 使用 X-Pack

X-Pack 插件包含安全、警报、监视、报告和图形功能。首先来安装 X-Pack 插件：

```
>bin/elasticsearch-plugin install x-pack
```

然后列出所有加载的插件：

```
>bin/elasticsearch-plugin list
```

可以看到，X-Pack 插件已经安装。如果不需要该插件，也可以删除，命令如下：

```
>bin/elasticsearch-plugin remove x-pack
```

为了和 Elasticstarch 通信，在使用 curl 命令的时候需要加入认证参数-user username:password，使用该参数访问索引：

```
#curl -user elastic:changeme -XPUT 'localhost:9200/idx'
```

7.1.3　快照

为了保证数据的完整性，可以使用快照功能把历史数据存入 Hadoop 集群的分布式文件系统（HDFS）上。HDFS 存储库插件增加了使用 HDFS 作为 Snapshot/Restore 存储库的支持。

HDFS 插件可以使用插件管理器安装。命令如下：

```
# bin/elasticsearch-plugin install repository-hdfs
```

HDFS 插件必须安装在集群中的每个节点上，每个节点必须在安装后重新启动。

HDFS 插件也可以从以下地址下载安装：

```
https://artifacts.elastic.co/downloads/elasticsearch-plugins/repository
-hdfs/repository-hdfs-5.3.0.zip
```

HDFS 快照/恢复插件是针对最新的 Apache Hadoop 2.x（目前版本为 2.7.1）构建的。如果读者正在使用的发行版与 Apache Hadoop 不兼容，可以考虑使用自己的插件文件替换插件文件夹中的 Hadoop 库。

安装后，通过 REST API 定义 HDFS 仓库的配置：

```
PUT /_snapshot/my_backup
{
"type": "hdfs",
"settings": {
     "path": "/path/on/hadoop",                          //Hadoop 集群上的地址
     "uri": "hdfs://hadoop_cluster_domain:[port]", //访问 Hadoop 的网址
     "conf_location":"/hadoop/hdfs-site.xml,/hadoop/core-site.xml",
                                                         //配置文件路径
     "user":"hadoop"                                     //访问用户
   }
}
```

为索引创建快照：

```
PUT /_snapshot/my_backup/snapshot_1?wait_for_completion=true
```

可以使用以下命令恢复快照：

```
POST /_snapshot/my_backup/snapshot_1/_restore
```

Lucene 索引可以增量备份，以便减少硬盘空间的占用。

7.2 Logstash 数据处理工具

Logstash 是一个收集、处理和转发应用程序事件和日志消息的工具软件，可以用它来统一对应用程序日志进行收集管理，提供 Web 接口用于查询和统计。

通过可配置的输入插件来完成数据收集，包括原始套接字/数据包通信、文件尾部追踪和几个消息总线客户端。一旦输入插件收集到数据后，就可以通过多个过滤器进行处理。

最后，Logstash 把处理过的事件传递给输出插件，这些插件可以将事件转发到各种外部程序中，如 Elasticsearch、本地文件和多个消息总线实现。

7.2.1 使用 Logstash

可以用 Yum 安装 Logstash。首先在/etc/yum.repos.d/路径下写一个 repo 文件指定软件包所在的网址。repo 文件是 Yum 源（软件仓库）的配置文件。logstash.repo 文件内容如下：

```
# vi /etc/yum.repos.d/logstash.repo
[logstash-5.x]                          #软件源的名称
name=Elastic repository for 5.x packages        #为了方便阅读配置文件用的名称
baseurl=https://artifacts.elastic.co/packages/5.x/yum
                                        //源的镜像服务器地址
gpgcheck=1                              #这个 repo 中下载的 rpm 将进行 gpg 的校验
gpgkey=https://artifacts.elastic.co/GPG-KEY-elasticsearch
                                        #定义用于校验的 gpg 密钥
enabled=1                               #这个 repo 中定义的源是启用的，0 为禁用
autorefresh=1                           #自动更新
type=rpm-md                             #指定存储库的类型
```

然后可以用 Yum 命令安装 Logstash：

```
# sudo yum install logstash
```

接下来，通过最简单的方法测试 Logstash。

Logstash 管道有两个必需元素，即输入和输出，以及一个可选元素过滤器。输入插件消耗来自源的数据，过滤器插件会按照指定的方式修改数据，输出插件将数据写入目的地。

然后到安装目录下运行最基本的 Logstash 管道：

```
# cd /usr/share/logstash/bin
# ./logstash -e 'input { stdin { } } output { stdout {} }'
```

在控制台输入 hello world，可以看到类似如下的输出：

```
2017-09-21T08:44:57.280Z iZetooc2z5ucu3Z hello world
```

可以在配置文件中指定输入和输出。一个简单的例子如下：

```
# vi logstash-simple.conf
input { stdin { } }
output {
  stdout { codec => rubydebug }                          //采用 Ruby 库来解析日志
}
```

使用这个配置文件启动 Logstash：

```
# ./logstash -f logstash-simple.conf
```

这里用到了 Codec。根据输入 test，产生的输出如下：

```
{
    "@timestamp" => 2017-08-21T10:18:20.749Z,          //时间戳
      "@version" => "1",                               //版本
          "host" => "iZetooc2z5ucu3Z",                 //主机名
       "message" => "test"                             //消息
}
```

数据可以输出到多个目的地，同时输出到 Elasticsearch 和 stdout 的配置如下：

```
input { stdin { } }
output {
  elasticsearch { hosts => ["localhost:9200"] }
  stdout { codec => rubydebug }
}
```

Linux 系统使用 Logstash，最简单的方法是使用传统的日志记录方法 Syslog。

Syslog 是计算机日志记录的原始标准之一，是由埃里克·阿尔曼（Eric Allman）设计的，是 Sendmail 的一部分，并且已经成长为支持各种日志记录的平台和应用程序。Syslog 已成为 Linux 系统上记录日志的默认机制，运行在 Linux 上的应用程序及打印机和网络设备，如路由器、交换机和防火墙等大多使用 Syslog。

Syslog 配置文件位于/etc/rsyslog.conf 下。默认情况下，RedHat/Fedora 的/etc/rsyslog.conf 文件被配置为将大多数消息放入文件/var/log/messages 下。Syslog 产生的每个消息的大致结构如下：

```
Aug 28 13:50:50 iZetooc2z5ucu3Z systemd: Starting Session 13738 of user root.
```

消息的组成部分包括：时间戳、生成消息的主机、生成消息的进程和消息的内容。

可以配置 Logstash 服务器接收 Syslog 消息。Logstash 配置文件位于/etc/logstash/conf.d 目录下。编辑/etc/logstash/conf.d/syslog.conf 文件：

```
input{
    syslog{
    type => "system-syslog"                              //指定输入类型
    port => 514                                          //系统日志服务监听 514 端口
    }

}

output{
```

```
    stdout{
        codec => rubydebug
    }

}
```

然后用指定配置文件启动 Logstash：

```
# ./logstash -f /etc/logstash/conf.d/syslog.conf
```

7.2.2 插件

可以通过程序 bin/logstash-plugin 来安装、删除和升级插件。例如，列出当前可用的插件：

```
# ./logstash-plugin list
```

也可以安装、更新或者删除插件。例如，安装 logstash-output-kafka 插件：

```
# ./logstash-plugin install logstash-output-kafka
```

更新 logstash-output-kafka 插件：

```
# ./logstash-plugin update logstash-output-kafka
```

删除 logstash-output-kafka 插件：

```
# ./logstash-plugin remove logstash-output-kafka
```

7.2.3 数据库输入插件

MongoDB 输入插件的配置文件如下：

```
input {
  mongodb {
    uri => 'mongodb://10.0.0.30/my-logs?ssl=true'   //连接到指定的数据库
    path => '/opt/logstash-mongodb/logstash_sqlite.db'
    collection => 'events_'                          //取得指定的文档集合
    batch_size => 5000                               //取得文档的数量
  }
}
```

这里的数据库文件'/opt/logstash-mongodb/logstash_sqlite.db'是自动创建出来的。

应该为/opt/logstash-mongodb 上的 Logstash 服务器提供权限，以便它能够创建数据库：

```
# chown -R logstash:logstash /opt/logstash-mongodb
```

可以使用 Logstash 同步 MySQL 数据到 Elasticsearch 中，不捕获表上的 DELETE 操作，它们只能捕获 INSERT 和 UPDATE 操作。

可以通过 JDBC 从 MySQL 读取数据。例如：

```
input {
  jdbc {
```

```
    jdbc_driver_library => "/path/to/mysql-connector-java-5.1.33-bin.jar"
                                                          //驱动程序
    jdbc_driver_class => "com.mysql.jdbc.Driver"          //驱动类
    jdbc_connection_string => "jdbc:mysql://host:port/database"
                                                          //连接字符串
    jdbc_user => "user"                                   //用户名
    jdbc_password => "password"                           //密码
    statement => "SELECT ..."                             //查询语句
    jdbc_paging_enabled => "true"                         //分页
    jdbc_page_size => "50000"                             //分页大小
  }
}

filter {
  [some filters here]
}

output {                                                  //两个输出目标
  stdout {
    codec => rubydebug
  }
  elasticsearch_http {
    host => "host"
    index => "myindex"
  }
}
```

7.2.4　开发插件

我们采用 Ruby 开发插件，因此需要安装 Ruby 管理工具 Bundler。Bundler 通过跟踪和安装所需的 gem（Ruby 模块）版本，为 Ruby 项目提供一致的环境。

首先安装 gem：

```
# yum install gem
```

然后安装 Bundler：

```
# gem install bundler
```

测试 Rspec 框架：

```
# bundle exec rspec
```

安装测试/规范所需的 Logstash 包在 Gemfile 中由类似以下的代码指定：

```
# gem "logstash", :github => "elasticsearch/logstash", :branch => "1.5"
```

7.3　Filebeat 文件收集器

Filebeat（下载地址是 https://github.com/elastic/beats/tree/master/filebeat）是一个开源文

件收集器，可以使用它获取日志文件并将其提供给 Logstash。目前，Filebeat 可以给 Elasticsearch 或者 Logstash 发送数据。

在 Linux 下，可以使用 RPM 安装 Filebeat。命令如下：

```
# curl -L -O https://artifacts.elastic.co/downloads/beats/filebeat/
filebeat-5.5.2-x86_64.rpm
# rpm -vi filebeat-5.5.2-x86_64.rpm
```

在配置文件/etc/filebeat/filebeat.yml 中指定了监测的日志路径和输出的目标。如果要输出到 Logstash 上，可以修改/etc/filebeat/filebeat.yml 的相关内容如下：

```
output:
  logstash:
    hosts: ["localhost:5044"]
```

在 Logstash 的配置文件 logstash.conf 中指定从端口（5044）监听来自 Filebeat 的数据。命令如下：

```
input {
beats {
port => '5044'
}
}
```

测试启动 Filebeat：

```
# /usr/share/filebeat/bin/filebeat -e -c /etc/filebeat/filebeat.yml -d
"publish"
```

默认的 Elasticsearch 需要的索引模板文件在安装 Filebeat 的时候已经提供，路径为/etc/filebeat/filebeat.template.json，可以使用如下命令装载该模板：

```
$ curl -XPUT
   'http://localhost:9200/_template/filebeat?pretty'
-d@/etc/filebeat/filebeat.template.json
```

为了收集 Tomcat 日志，可以修改配置文件/etc/filebeat/filebeat.yml：

```
filebeat.prospectors:

- input_type: log

  paths:
- /install/tomcat/logs/catalina.out
```

设置 Filebeat 服务，让其在服务器重新启动时自动启动，命令如下：

```
# systemctl enable filebeat
# systemctl start filebeat
```

7.4 消息过期

可以使用 Elasticsearch 的 TTL（Time-to-Live）功能定期删除过期的日志信息。

为了让 TTL 工作，首先必须在映射中启用 TTL（默认情况下是禁用的），然后在索引文档时设置 TTL 值。示例如下：

```
# curl -XPUT 'mybox:9200/blog/user/_mapping?pretty' -d '{
  "user": {
     "_ttl": {"enabled": true}
}'

# curl -XPUT 'mybox:9200/blog/user/dilbert' -d '{ "name" : "Dilbert Brown",
"_ttl": "3m"}'

# curl -XGET 'mybox:9200/blog/user/dilbert?pretty'
```

7.5　Kibana 可视化平台

Kibana 是一个用于 Elasticearsh 的开源分析与可视化平台，下载网址为 https://github.com/elastic/kibana。本节将介绍如何安装和使用 Kibana。

首先导入 Elastic 栈的 PGP Key：

```
# rpm --import https://artifacts.elastic.co/GPG-KEY-elasticsearch
```

然后用 Micro 创建新的 repo 文件：

```
# micro /etc/yum.repos.d/kibana.repo
```

repo 文件内容如下：

```
[kibana-5.x]
name=Kibana repository for 5.x packages
baseurl=https://artifacts.elastic.co/packages/5.x/yum
gpgcheck=1
gpgkey=https://artifacts.elastic.co/GPG-KEY-elasticsearch
enabled=1
autorefresh=1
type=rpm-md
```

和 Logstash 一样，可以用 Yum 安装 Kibana，命令如下：

```
# yum install kibana
```

然后启动 Kibana：

```
# systemctl start kibana
```

可以用字符浏览器 Links 在本机查看启动状态：

```
# links http://localhost:5601/status
```

为了能够远程访问 Kibana，修改 Kibana.yml 文件如下：

```
# micro /etc/kibana/kibana.yml
```

然后修改 IP 地址：

```
server.host: "0.0.0.0"
```

重新启动 Kibana 服务，让配置生效：

```
# systemctl restart kibana
```

这样就可以通过远程访问 Kibana 服务了。

在配置文件 Kibana.yml 中增加 Kibana 查询的 Elasticsearch 地址如下：

```
elasticsearch.url: http://localhost:9200
```

在开始使用 Kibana 之前，可以先定义一个 index pattern 用来匹配一个或多个索引，告诉 Kibana 探索哪个 ElasticSearch 索引。

7.6 Flume 日志收集系统

Apache Flume（http://flume.apache.org/）是一种分布式、可靠和可用的服务，用于高效收集、聚合和移动大量日志数据。它具有基于流数据流的简单灵活的架构。

Flume 事件被定义为具有字节有效载荷和可选的一组字符串属性的数据流的单元。Flume 代理是一个 JVM 进程，它承载事件从外部源（source）传递到下一个目标（sink）的组件。可以使用 ElasticSearchSink 将 Flume 采集的数据传输到 Elasticsearch 中。

下载 Flume 安装包：

```
# wget http://mirror.bit.edu.cn/apache/flume/1.8.0/apache-flume-
1.8.0-bin.tar.gz
```

然后解压缩：

```
# tar -zxvf apache-flume-1.8.0-bin.tar.gz
```

使用位于 Flume 安装目录 bin 中的名为 flume-ng 的 shell 脚本启动代理，需要在命令行中指定代理名称、config 目录和配置文件：

```
$ bin/flume-ng agent -n $agent_name -c conf -f conf/flume-conf.properties.
template
```

默认情况下，Flume 将一行视为一个事件。例如，通过 tail 命令监控 Tomcat 日志文件，配置文件内容如下：

```
agent.sources.SrcLog.type = exec
agent.sources.SrcLog.command = tail -F /home/tomcat/webapps/logs/
catalina.out
agent.sources.SrcLog.restart = true
agent.sources.SrcLog.restartThrottle = 1000
agent.sources.SrcLog.logStdErr = true
agent.sources.SrcLog.batchSize = 50
```

Spooling Directory Source 监视指定的文件夹下有没有写入新的文件，如果有的话，就会把该文件内容传递给目标组，然后将该文件名后缀标示为.complete，表示已处理。Spooling Directory Source 提供了一些参数可以将文件名和文件全路径名添加到事件的 header 中。

ElasticSearchSink 配置如下：

```
agent.sinks.elasticSearchSink.type
    = org.apache.flume.sink.elasticsearch.ElasticSearchSink
agent.sinks.elasticSearchSink.channel = fileChannel
agent.sinks.elasticSearchSink.hostNames=localhost:9300
agent.sinks.elasticSearchSink.indexName=platform
agent.sinks.elasticSearchSink.indexType=platformtype
agent.sinks.elasticSearchSink.ttl=1m
agent.sinks.elasticSearchSink.batchSize=1000
agent.sinks.elasticSearchSink.serializer=
org.apache.flume.sink.elasticsearch.ElasticSearchLogStashEventSerializer
```

7.7　Kafka 分布式流平台

Apache Kafka 最初是一个用于日志处理的分布式消息队列，同时支持离线和在线日志处理。Kafka 在消息保存时能根据主题对消息进行归类。

通过在 Kafka 连接器，在 Kafka 和另一个系统之间复制数据，用户可以为要从中提取数据的系统实例化 Kafka 连接器，也可以将数据推送到 Kafka 连接器上。源连接器将数据从另一个系统（如关系数据库）导入数据。Sink 连接器导出数据（例如将 Kafka 主题的内容导出到 Elasticsearch 或者 HDFS 文件系统）。

Kafka 后来发展成一个分布式流平台。流平台有 3 个关键功能：

- 允许发布和订阅记录流，类似于消息队列或企业消息系统。
- 允许以容错方式存储记录流。
- 可以处理记录流。

一个典型的 Kafka 集群中包含若干生产者（可以是 Web 前端产生的网页浏览记录或者是服务器日志等）、若干代理（Kafka 支持水平扩展，一般代理数量越多，集群吞吐率越高）、若干消费者集群，以及一个 Zookeeper 集群。Kafka 通过 Zookeeper 管理集群配置，选举 leader，在消费者集群发生变化时进行重新平衡。生产者使用 push 模式将消息发布给代理，消费者使用 pull 模式通过代理订阅并消费消息。

如果日志写入 Elasticsearch 是系统瓶颈，那么可以考虑使用 Logstash 从 Kafka 中获取数据并将其推送到 Elasticsearch 上。Logstash 配置文件内容如下：

```
input {
  kafka {
    bootstrap_servers => "localhost:9092"// Bootstrap 服务器与 Kafka 代理相同
    topics => ["beats"]                  //主题为 beats
  }
}
output {
  elasticsearch {
    hosts => ["localhost:9200"]
    index => "elasticse"
```

```
  }
}
```

可根据代理配置定期删除过期数据。例如：

```
log.cleanup.policy=delete                    //启用删除策略，也可以使用压缩策略
```

如直接删除的话，删除后的消息不可恢复。可按时间清理，例如：

```
log.retention.hours=16                       //超过指定时间则清理
```

或者按存储容量来清理：

```
log.retention.bytes=1073741824               //超过指定容量后，删除旧的消息
```

7.8 Graylog 日志管理平台

Graylog（https://www.graylog.org/）是一个开源日志管理平台，使用 Elasticsearch 存储日志消息，MongoDB 存储元信息和配置数据。

Graylog Collector Sidecar（https://github.com/Graylog2/collector-sidecar）是第三方日志收集器（如 NXLog）的主管流程。Sidecar 程序能够从 Graylog 服务器上获取配置，并将它们转换为各种日志收集器的有效配置文件。可以将它当作对于日志收集器的集中式配置管理系统。

为了在 Linux 下安装 Graylog 服务器，首先需要安装 MongoDB 和 Elasticsearch。下载 MongoDB：

```
curl -O https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-3.4.4.tgz
```

解压缩：

```
# tar -zxvf mongodb-linux-x86_64-3.4.4.tgz
```

在首次启动 MongoDB 之前，需要创建 mongod 进程写入数据的目录。默认情况下，mongod 会将数据写入/data/db 目录。使用以下命令创建该目录：

```
# mkdir -p /data/db
```

运行可执行文件 mongod：

```
<path to binary>/mongod
```

创建服务：

```
# mkdir ../log/
# ./mongod --fork --logpath ../log/mongod.log
```

Graylog 2.3 支持 Elasticsearch 5，所以我们安装 Elasticsearch 5.6 版本。

首先导入用来验证 RPM 包的 RPM GPG 公钥：

```
# rpm --import https://packages.elastic.co/GPG-KEY-elasticsearch
```

然后添加 Elasticsearch 仓库：

```
# vi /etc/yum.repos.d/elasticsearch.repo

[elasticsearch-5.6]
name=Elasticsearch repository for 5.6.x packages
baseurl=http://packages.elastic.co/elasticsearch/5.6/centos
gpgcheck=1
gpgkey=http://packages.elastic.co/GPG-KEY-elasticsearch
enabled=1
```

使用 Yum 命令安装 Elasticsearch 5.6 版本：

```
# yum -y install elasticsearch
```

配置 Elasticsearch 在系统启动的过程中启动：

```
# systemctl daemon-reload
# systemctl enable elasticsearch.service
```

唯一重要的是将集群名称设置为 graylog，这是由 Graylog 使用的。现在编辑 Elasticsearch 的配置文件如下：

```
# vi /etc/elasticsearch/elasticsearch.yml
cluster.name: graylog
```

还需要禁用动态脚本以避免远程执行，可以通过在上述文件末尾添加以下命令行来完成。

```
script.disable_dynamic: true
```

一旦完成，就可以重新启动 Elasticsearch 服务以加载修改的配置。命令如下：

```
# systemctl restart elasticsearch.service
```

等待至少一分钟让 Elasticsearch 完全重新启动，否则测试将失败。Elastisearch 现在应该监听 9200 端口处理 HTTP 请求，可以使用 CURL 命令来获取响应，确保它返回的集群名称为 graylog2。

```
# curl -X GET http://localhost:9200

{
  "status" : 200,
  "name" : "Silver Fox",
  "cluster_name" : "graylog",                                    //集群名称
  "version" : {
    "number" : "5.6.2",
    "build_hash" : "e43676b1385b8125d647f593f7202acbd816e8ec",
    "build_timestamp" : "2015-09-14T09:49:53Z",
    "build_snapshot" : false,
    "lucene_version" : "4.10.4"
  },
  "tagline" : "You Know, for Search"
}
```

使用以下命令检查 Elasticsearch 集群的运行状况，必须让集群的状态是 green，以使 Graylog 正常工作。

```
# curl -XGET 'http://localhost:9200/_cluster/health?pretty=true'

{
```

```
  "cluster_name" : "graylog",
  "status" : "green",                                    //集群健康状况
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 0,
  "active_shards" : 0,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0,
  "delayed_unassigned_shards" : 0,
  "number_of_pending_tasks" : 0,
  "number_of_in_flight_fetch" : 0
}
```

接下来安装 Graylog 2。Graylog-server 接收并处理日志消息，并为来自 graylog-web-interface 的请求产生 REST API。从 graylog.org 可以下载最新版本的 Graylog。

使用如下命令安装 Graylog 2 存储库。

```
#rpm -Uvh
  https://packages.graylog2.org/repo/packages/graylog-2.3-repository_
  latest.rpm
```

安装最新的 Graylog 服务器：

```
# yum -y install graylog-server
```

编辑 server.conf 文件：

```
# vi /etc/graylog/server/server.conf
```

在上述文件中配置以下变量。

首先设置一个秘钥来保护用户密码，可以使用以下命令生成一个秘钥，至少使用 64 个字符。

```
# pwgen -N 1 -s 96
5uxJaeL4vgP9uKQ1VFdbS5hpAXMXLq0KDvRgARmlI7oxKWQbH9tElSSKTzxmj4PUGlHIpOk
oMMwjICYZubUGc9we5tY1FjLB
```

如果还没有安装 pwgen，可以使用以下命令安装 pwgen。

```
# yum -y install pwgen
```

然后放置秘钥：

```
password_secret = 5uxJaeL4vgP9uKQ1VFdbS5hpAXMXLq0KDvRgARmlI7oxKWQbH9
tElSSKTzxmj4PUGlHIpOkoMMwjICYZubUGc9we5tY1FjLB
```

接下来为 root 用户设置散列密码(不要与系统用户混淆，Graylog 的 root 用户是 admin)，然后使用设置的密码登录到 Web 界面，管理员的密码不能在 Web 界面更改，必须在后台修改配置文件后才能更改。

用用户选择的密码替换 yourpassword。

```
# echo -n yourpassword | sha256sum
e3c652f0ba0b4801205814f8b6bc49672c4c74e25b497770bb89b22cdeb4e951
```

然后放置散列密码：

```
root_password_sha2 =
    e3c652f0ba0b4801205814f8b6bc49672c4c74e25b497770bb89b22cdeb4e951
```

还可以设置电子邮件地址 root（admin）用户：

```
root_email = "itzgeek.web@gmail.com"
```

然后设置 root（admin）用户的时区：

```
root_timezone = UTC
```

Graylog 可以尝试自动查找 Elasticsearch 节点，它使用组播模式。但是对于较大的网络，建议使用最适合生产设置的单播模式。因此，将以下两个条目添加到 graylog server.conf 文件中，将 ipaddress 替换为 live hostname 或 ipaddress，可以使用逗号分隔多个主机。命令如下：

```
elasticsearch_http_enabled = false
elasticsearch_discovery_zen_ping_unicast_hosts = ipaddress:9300
```

通过定义以下变量设置唯一的一个主节点，默认设置为 true，但必须将其设置为 false，以使特定节点作为从属节点。主节点执行一些从属节点不执行的周期性任务。

```
is_master = true
```

以下变量设置每个索引保留的日志消息数，建议使用几个较小的索引，而不是较大的索引。

```
elasticsearch_max_docs_per_index = 20000000
```

以下参数定义索引的总数，如果此数字达到，旧索引将被删除。

```
elasticsearch_max_number_of_indices = 20
```

分片设置取决于 Elasticsearch 集群中的节点数，如果只有一个节点，则将其设置为 1。例如：

```
elasticsearch_shards = 1
```

然后设置索引的副本数，如果 Elasticsearch 集群中只有一个节点，将其设置为 0。例如：

```
elasticsearch_replicas = 0
```

添加 MongoDB 身份验证信息如下：

```
mongodb_useauth = false
```

使用以下命令启动 Graylog 服务器。

```
# systemctl restart graylog-server
```

还可以查看服务器启动日志，如果出现任何问题，则会很有用。

```
# tailf /var/log/graylog-server/server.log
```

在成功启动 graylog-server 后，应该在日志文件中收到以下消息。

```
2015-09-16T21:26:05.689-04:00 INFO  [ServerBootstrap] Graylog server up
and running.
```

下面安装 Graylog 网页界面。

要配置 graylog-web 界面，至少要有一个 graylog-server 节点。使用以下命令安装 Web 界面。

```
# yum -y install graylog-web
```

然后编辑配置文件并设置以下参数：

```
# vi /etc/graylog/web/web.conf
```

以下是 graylog-server 节点的列表，还可以添加多个节点，用逗号分隔。

```
graylog2-server.uris="http://127.0.0.1:12900/"
```

设置应用程序 scret，可以使用命令 pwgen -N 1 -s 96 来生成。

```
application.secret="sNXyFf6B4Au3GqSlZwq7En86xp10JimdxxYiLtpptOejX6tIUpU
E4DGRJOrcMj07wcK0wugPaapvzEzCYinEWj7BOtHXVl5Z"
```

使用以下命令重新启动 gralog-web 界面：

```
# systemctl restart graylog-web
```

访问 Graylog Web 界面：

Web 界面将侦听端口 9000，配置防火墙以允许端口 9000 上的流量。

```
# firewall-cmd --permanent --zone=public --add-port=9000/tcp
# firewall-cmd --reload
```

将浏览器指向 http://ip-add-ress:9000，然后使用用户名 admin 和在 server.conf 文件中 root_password_sha2 选项指定的密码登录。

7.9 本章小结

本章主要介绍了 Elastic 栈日志监控与分析系统的集群搭建与使用。其中重点介绍了管理 Elasticsearch 集群的权限控制方法及数据备份与恢复的方法，可以使用 Nginx 或者 X-Pack 插件实现 Elasticsearch 写入权限控制。为了实现数据备份，可以搭建 Hadoop 平台，并存储 ELK 的历史数据。

Elasticsearch 与数据收集和日志解析引擎 Logstash、分析和可视化平台 Kibana 一起开发。这三款产品被设计成集成解决方案，被称为“Elastic 栈”（以前称为“ELK 栈”）。

Logstash 事件处理管道有三个阶段：input→filter→output。Logstash 早期的版本中，input、filter、output 分别设置为独立的线程，连接成一个管道。但这种架构方式会导致数据在管道中多次被复制，造成性能损耗。从 Logstash 2.3 开始，filter 和 output 共用一个管道线程。

Codec 是从 Logstash 1.3.0 开始引入的概念（Codec 来自 Coder/Decoder 两个单词的缩写），其可以作为一部分输入或输出操作的流过滤器。

用户通过 Mailgun 收发大量电子邮件，而 Mailgun 跟踪和存储每封邮件发生的每个事件。每个月会新增数十亿事件，需要展示给客户，方便他们进行数据分析，也就是全文搜索，利用 Elasticsearch 和 Logstash 技术可以完成这个需求。

Elasticsearch 2.0 以前的版本使用 Elasticsearch river 同步数据库中的数据到 Elasticsearch。因为 Elasticsearch river 在 Elasticsearch 进程内部运行，运行时需要额外的内存、更多的套接字、文件描述符等，这样可能导致集群不稳定。因此当 Elasticsearch river 被废弃后，一些 Elasticsearch river 应用后来被实现为 Logstash 插件。

Logstash 插件的一些文档是自动生成的，源代码中的注释将首先转换为方便编辑的 Asciidoc 文件，然后转换为 HTML 文件。

Flume 是分布式的日志收集系统，其具有很好的容错能力、可调节的可靠性机制和许多故障转移和恢复机制。

Graylog 服务器采用 Java 开发，是一个开源日志管理平台，也使用 Elasticsearch 进行存储和搜索日志。可以使用 Elasticsearch、Graylog 和 MongoDB 构建分布式日志服务器。

另外，向读者推荐几个通用的数据浏览器，如 DejaVu（网址为 https://github.com/appbaseio/dejavu）、Cerebro（网址为 https://github.com/lmenezes/cerebro/），以及 ElasticHQ（网址为 https://github.com/ElasticHQ/elasticsearch-HQ）。其中，Dejavu 是 Chrome 的扩展，Cerebro 是使用 Scala、Play Framework、AngularJS 和 Bootstrap 构建的开源 Elasticsearch Web 管理工具。

Elasticsearch 系统监控为随后的行为引导和系统进化提供了可能性。

第 8 章　案例分析

本章首先介绍有助于提高自然语言理解能力的双语句对搜索，然后介绍网站内容管理系统 dotCMS 及其使用的 Elasticsearch 站内搜索。

8.1　双语句对搜索

提高对自然语言的理解能力能够提高搜索的准确度。为了让分词更准确，可以挖掘和利用一些富含知识的数据。双语句对搜索往往很准确，而且包含丰富的语言信息，可以作为数据挖掘的起点。

8.1.1　爬虫抓取双语句对

我们可以使用 OkHttp（https://github.com/square/okhttp）下载双语句对。在项目中引入两个 jar 包：okio-1.13.0.jar 和 okhttp-3.9.1.jar。用爬虫抓取必应词典的代码如下：

```
public class GetExample {
  //使用系统默认的参数来创建 OkHttpClient 对象
  OkHttpClient client = new OkHttpClient();

  String run(String url) throws IOException {
    //创建一个请求
    Request request = new Request.Builder()
        .url(url)
        .build();
    //同步阻塞调用
    try (Response response = client.newCall(request).execute()) {
      return response.body().string();
    }
  }

  public static void main(String[] args) throws IOException {
    GetExample example = new GetExample();
    //按词条返回例句
    String response = example.run("http://cn.bing.com/dict/search?
    q=windowsill");
    System.out.println(response);                    //输出抓取的网页结果
  }
}
```

使用 Jsoup 解析网页内容，提取英文和中文翻译对照句子。

```
Document doc = Jsoup.parse(content);
Elements elementsLi = doc.select(".se_li");     //根据CSS的class类型选择元素

for (Element pairs : elementsLi) {
Element s = pairs.select(".se_li1").first();    //选取第一个 se_li1 样式的元素
//选取第一个 sen_en 样式元素的文本
String senEn = s.select(".sen_en").first().text();
//选取第一个 sen_cn 样式元素的文本
String senCn = s.select(".sen_cn").first().text();

System.out.println(senCn);                      //输出中文句子
System.out.println(senEn);                      //输出英文句子
}
```

8.1.2　英文分词

如果未指定所使用的分析器，那么默认使用的是 Standard Analyzer，可以切分出英文分词。为了更深入地分析英文文章，可以专门开发针对英文的分析器。

8.1.3　句子切分

句子切分并不是一个简单的问题。标点符号“?”和“!”的含义比较单一。但是“.”有很多种不同的用法，并不一定是句子的结尾。例如，“Mr. Vinken is chairman of Elsevier N.V., the Dutch publishing group. ”需要排除掉一部分情况，如果“.”是某个短语中间的一部分，则它不是句子的结尾。这里的 Mr. Vinken 是一个人名短语，如果这个人名正好不在词典中，则可以根据上下文识别规则识别出这个短语。示例如下：

```
//输入文本
String text= "Mr. Vinken is chairman of Elsevier N.V., the Dutch publishing
group.";
//构建英文文档类
EnText enText = new EnText(text);
for(Sentence sent:enText){                      //遍历文档中的句子
    System.out.println(sent);  //因为输入是一个句子，所以这里只会打印出一个句子
}
```

Java 中的 BreakIterator 类已经包含了切分句子的功能，可以用它实现一个英文句子迭代器。代码如下：

```
// SentBreakIterator 实现 Iterator 接口
private final static class SentBreakIterator implements Iterator<Sentence>
{
String text;                                    //文本
int start;                                      //开始位置
int end;                                        //结束位置
```

```
// 根据英文标点符号切分
static final BreakIterator boundary = BreakIterator
        .getSentenceInstance(Locale.ENGLISH);

public SentBreakIterator(String t) {
    text = t;
    //设置要处理的文本
    boundary.setText(text);
    start = boundary.first();                                    //开始位置
    end = boundary.next();
} // 用于迭代的类
@Override
public boolean hasNext() {                                       //看迭代是否已经结束
    return (end != BreakIterator.DONE);
}

@Override
public Sentence next() {
    String sent = text.substring(start, end);                    //切分出句子

    Sentence sentence = new Sentence(sent, start, end);          //构建句子对象
    start = end;
    end = boundary.next();
    return sentence;
}
}
```

BreakIterator 切分得不太准确，所以我们自己写一个句子切分器。输入当前切分点，找下一个切分点的代码如下：

```
public static int nextPoint(String text, int lastEOS) {
int i = lastEOS;
while (i < text.length()) {
    //跳过短语
    i = skipPhrase(text, i);

    //然后再找标点符号
    String toFind = eosDic.matchLong(text, i);                   //匹配标点符号词典
    if (toFind != null) {
        //判断是否是有效的可切分点。例如，在括号中的标点符号不是有效的可切分点
        boolean isEndPoint = isSplitPoint(text, lastEOS, i);
        if (isEndPoint) {
            return i + toFind.length();
        }
        i = i + toFind.length();
    } else {                                                     //没找到
        i++;
    }
}
return text.length();                                            //返回最大长度
}
```

SentIterator 是一个用于迭代英文文本返回句子的内部类，实现代码如下：

```
private final static class SentIterator implements Iterator<Sentence> {
                                                    //实现迭代器
String text;
int lastEOS = 0;                                    //返回当前处理到的位置

public SentIterator(String t) {
    text = t;
}

@Override
public boolean hasNext() {                          //看是否还有下一个句子
    return (lastEOS < text.length());
}

@Override
public Sentence next() {                            //返回下一个句子
    int nextEOS = EnSentenceSpliter.nextPoint(text, lastEOS);
                                                    //得到句子结束位置
    String sent = text.substring(lastEOS, nextEOS);
    Sentence sentence = new Sentence(sent, lastEOS, nextEOS); //构造出句子
    lastEOS = nextEOS;
    return sentence;                                //返回句子
}
}
```

8.1.4　标注词性

采集下来的英文句子可以标注其中单词的词性。例如一段英文：

Cats never fail to fascinate human beings. They can be friendly and affectionate towards humans, but they lead mysterious lives of their own as well.

标注词性后的结果是：

```
Cats(n.) never fail(v.) to(prep) fascinate(v.) human(n.) beings(n.).
They(pron.) can(aux.) be(v.) friendly(adj.) and(conj.) affectionate(adj.)
towards(prep) humans(n.) but(conj.) they(n.) lead(v.) mysterious(adj.)
lives(n.) of(prep.) their(n.) own(n.) as well(adv.).
```

这里用编码来表示词性，括号中的输出是词性编码。有些汉语中的量词是英语中没有的，如件、个、艘。而英语中也有一些独有的词性，如冠词 a、an、the。英文词性编码表如表 8-1 所示。

表 8-1　英文词性编码表

代　码	名　称
n.	名词
adj.	形容词
adv.	副词
art.	冠词

（续）

代　码	名　称
pos.	所有格
pron.	代词
aux.	情态助动词
conj.	连接词
v.	动词
num.	数词
prep.	介词
punct.	标点符号
int.	感叹词

词性标注的流程图如图 8-1 所示。

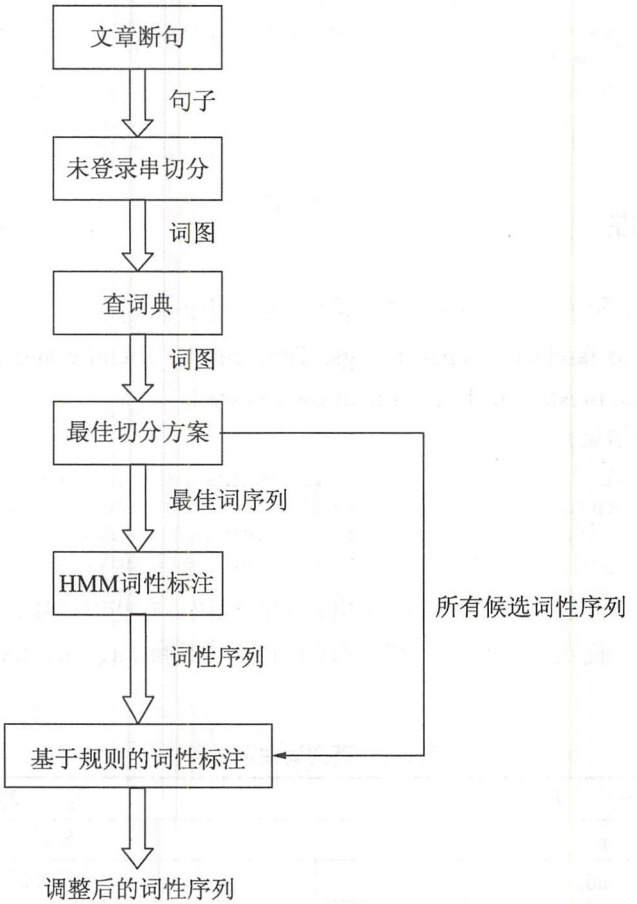


图 8-1　英文分词流程图

和中文词性标注一样，可以使用隐马可夫模型进行英文词性标注。英文词性标注语料库和中文词性标注语料库不一样，可以从 GitHub 网站上找到一些免费的词性标准语料库。

下面来看标注规则，例如 I like it 对应的词性序列为[pron v pron]。

```
key = new ArrayList<PartOfSpeech>();
key.add(PartOfSpeech.pron);                                    //I
key.add(PartOfSpeech.v);                                       //like
key.add(PartOfSpeech.pron);                                    //it
posTrie.addProduct(key);
```

实现代码如下：

```
public static ArrayList<WordToken> getWords(Sentence sent){
ArrayList<WordTokenInf> words = Segmenter.seg(sent);          //先分词
WordType[] tags = g.tag(words);                                //标注词性
//再把词性和词本身结合起来，返回完整的词性标注结果
int i=0;
ArrayList<WordToken> tokens = new ArrayList<WordToken>();

for(WordTokenInf w:words){
    WordToken t = new WordToken(w.baseForm,w.termText,w.start,w.end,
    tags[i]);
    ++i;
    tokens.add(t);
}

return tokens;
}
```

8.1.5　词对齐

可以从对齐语料库中挖掘双语词典及词之间的搭配关系。找出中、英文对照词表的方法，又叫做词对齐（word alignment）。最简单的词对齐就是从前往后逐个词地对应。例如“北京 地铁”翻译成 Beijing Subway，其中前后两个词是按顺序对齐的。

```
HashMap<String,String> wordMap = new HashMap<String,String>();
                                                    //词语对照表

String enSent = "Beijing Subway";                   //英语句子
String cnSent = "北京 地铁";                          //中文句子

StringTokenizer enTokenizer = new StringTokenizer(enSent);
                                                    //用空格分割英文句子
StringTokenizer cnTokenizer = new StringTokenizer(cnSent);
                                                    //用空格分割中文句子
while(enTokenizer.hasMoreElements()){               //有更多的词没遍历完
wordMap.put(cnTokenizer.nextToken(),enTokenizer.nextToken());
}
```

```
for(Entry<String, String> e:wordMap.entrySet()){          //输出对照词表
System.out.println(e.getKey()+":"+e.getValue());
}
```

词对齐的结果是两个词之间的双向图，有一条有向线段连接两个单词，当且仅当它们是彼此的翻译。一个词对齐的例子如图 8-2 所示。

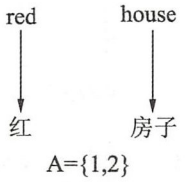


图 8-2　词对齐

这个对齐的概率就是 p(红| red)和 p(房子| house)的乘积。汉语句子的联合概率及其对应的英语对齐调整是所有这些概率的乘积。形式化的写法如下：

$$P(A,F \mid E) = \prod_{j=1}^{J} t(f_j \mid e_{a_j})$$

假设训练语料库中有两个对齐的句子：

红房子　　这房子

red house　　the house

红房子和 red house 的两种对齐方式如图 8-3 所示。

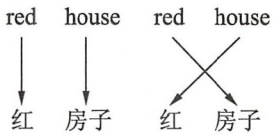


图 8-3 红房子和 red house 两种可能的对齐方式

第一种对齐方式的写法是 $A = \{1,2\}$；第二种对齐方式的写法是 $A = \{2,1\}$。

选择一个概率最大的对齐方式：

$$\hat{A} = \underset{A}{\operatorname{argmax}}\, P(F, A \mid E)$$

对某个位置来说：

$$a_j = \underset{1 \leqslant i \leqslant I}{\operatorname{argmax}}\, t(f_j, e_i) \quad 1 \leqslant j \leqslant J$$

- E 阶段：使用当前的翻译概率计算训练数据所有可能对齐的概率；
- M 阶段：使用这些对齐概率的估计去重新估计所有的翻译概率。

重复 E 和 M 阶段直到翻译概率值收敛为止。

- 初始阶段：

- 所有的词按相等可能对齐；
- 所有的 P(中文单词 | 英文单词)都相同。

例如，这里有 3 个英文单词{red,house,the}，3 个中文单词{红,房子,这}。

翻译概率如表 8-2 所示。

表 8-2　翻译概率表

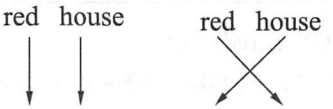
英文 \ 中文	红	房　子	这
red	1/3	1/3	1/3
house	1/3	1/3	1/3
the	1/3	1/3	1/3

例如，$t(\text{红}|\text{red})=\frac{1}{3}$。

计算对齐概率 $P(A,F\mid E)$，就是翻译概率的连乘积。计算 $P(A,F\mid E)$的公式如下：

$$P(A,F\mid E)=\prod_{j=1}^{J}t(f_j|e_{a_j})$$

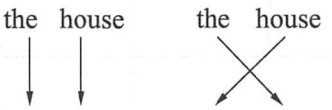
把所有句子的对齐概率都算一遍。例如，句子 red house 的对齐概率如下：



$$\frac{1}{3}\times\frac{1}{3}=\frac{1}{9}\qquad\frac{1}{3}\times\frac{1}{3}=\frac{1}{9}$$

$P(A,F|\text{"}red\ house\text{"})$ 的对齐概率

句子 the house 的对齐概率如下：

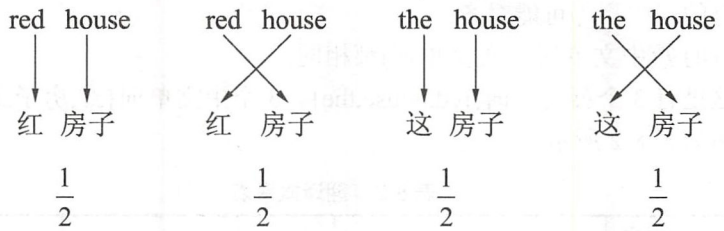


$$\frac{1}{3}\times\frac{1}{3}=\frac{1}{9}\qquad\frac{1}{3}\times\frac{1}{3}=\frac{1}{9}$$

$P(A,F|\text{"}the\ house\text{"})$ 的对齐概率

归一化，得到 $P(A\mid F,E)$：

$$\frac{1}{9}\times\frac{9}{2}=\frac{1}{2}$$



M 阶段根据 $P(A \mid F, E)$计算词的翻译概率。计算加权翻译计数 $C(fj,ea(j))$ += $P(a|e,f)$。得到的翻译概率如表 8-3 所示。

表 8-3　更新后的翻译概率

英文 \ 中文	红	房子	这
red	$\frac{1}{2}$	$\frac{1}{2}$	0
house	$\frac{1}{2}$	$\frac{1}{2}+\frac{1}{2}$	$\frac{1}{2}$
the	0	$\frac{1}{2}$	$\frac{1}{2}$

例如，C(红,red)=0.5、C(房子,red)=0.5。

归一化行，按行加的值是 1，得到翻译概率如表 8-4 所示。

表 8-4　再次更新后的翻译概率

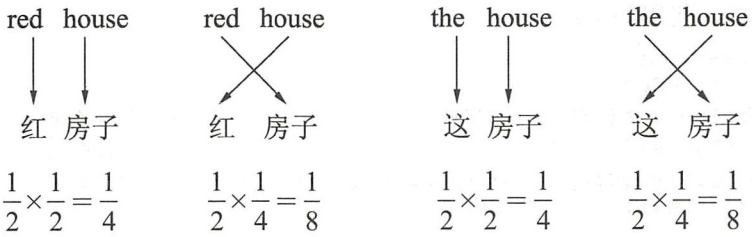
英文 \ 中文	红	房子	这
red	$\frac{1}{2}$	$\frac{1}{2}$	0
house	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{4}$
the	0	$\frac{1}{2}$	$\frac{1}{2}$

例如，t(红|red)=0.5 t(房子|red)=0.5

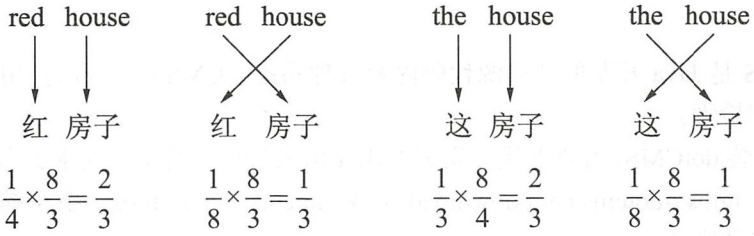
根据翻译概率乘积来估计 $P(f \mid e)$：

$$P(A,F \mid E)=\prod_{j=1}^{J} t(f_j \mid e_{a_j})$$

重新计算对齐概率 $P(A, F \mid E)$：



可以归一化进一步得到 $P(A,F\mid E)$的准确值 $P(A|E,F)=\dfrac{P(A,F|E)}{\sum_A P(A,F|E)}$



继续进行 EM 迭代，直到翻译参数收敛为止。相关实现代码可参考 Berkeley Aligner（见网址 https://github.com/mhajiloo/berkeleyaligner）。

8.1.6　索引数据

定义索引库结构，代码如下：

```
{
  "mappings": {
   "ensent": {                                    //英文句子
       "type":     "string",
       "analyzer": "english"
    },
   "cnsent": {                                    //中文句子
       "type":     "string",
       "analyzer": "cn_analyzer"
    }
}
```

放入索引数据，代码如下：

```
String id="2";                                    //唯一列的值
IndexRequestBuilder indexRequestBuilder = client.prepareIndex(
    "corpus", "encn", id);
Map<String, String> source = new HashMap<>();
source.put("ensent", "We wandered down the block and sat down to rest on
a windowsill.");
source.put("cnsent", "我们漫步走过这片街区，然后在一排窗台前坐下休息。");

indexRequestBuilder.setSource(source);
IndexResponse response = indexRequestBuilder.execute().actionGet();
```

查询数据，代码如下：

```
SearchResponse response = client
    .prepareSearch()
    .setIndices(INDEX_NAME)                    //设置索引名为 corpus
    .setTypes("encn")                          //设置索引类型为 encn
    .setQuery(QueryBuilders.matchQuery("ensent", "windowsill"))//查询词
    .execute().actionGet();
```

8.2 内容管理系统站内检索

dotCMS 是 Java 开发的开放源代码内容管理系统（CMS），可以使用 REST API 进行存储/更新和检索。

为了安装 dotCMS，需要下载并安装 JDK 1.8，安装的文件夹名称和路径不能包含空格。

可以从 https://dotcms.com/download/下载 dotcms_4.1.1.tar.gz。解压缩后在 Linux/OS-X/Unix 下启动服务：

```
./bin/startup.sh
```

这样实际上就是启动了一个 Tomcat 实例。

8.2.1 MySQL 数据库

虽然 dotCMS 积极地缓存并会尝试限制网站前端的数据库流量，但是数据库仍然是整个 dotCMS 系统的重要组成部分，数据库性能对 dotCMS 是至关重要的，尤其是在创作环境下。

dotCMS 默认使用的数据库是 H2，可以更改为 MySQL。方法是：打开 tomcat-8.0.18\webapps\ROOT\META-INF\context.xml，将 MySQL 部分的注释去掉，将 H2 注释掉，同时配置需要连接的数据库名称、用户名和密码。代码如下：

```
<Resource name="jdbc/dotCMSPool" auth="Container"
      type="javax.sql.DataSource" driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://localhost/dotcms?characterEncoding=UTF-8"
      username="{your db user}" password="{your db password}" maxTotal="60"
      maxIdle="10" maxWaitMillis="60000"
      removeAbandonedOnBorrow="true" removeAbandonedOnMaintenance="true"
      removeAbandonedTimeout="60" logAbandoned="true"
      timeBetweenEvictionRunsMillis="30000" validationQuery="SELECT 1"
      testOnBorrow="true" testWhileIdle="true" />
```

强烈建议对 context.xml 文件进行的所有更改都通过插件中的 ROOT 文件夹来操作。

创建 dotCMS 表时，请确保使用 UTF-8 字符集和 DEFAULT UTF-8 排序规则来创建。例如：

```
create database dotcms_zip default character set = utf8 default collate =
utf8_general_ci;
```

如果 MySQL 启用了二进制日志，则将以下内容放入 my.cnf 文件（对于 UNIX/Mac OS）或 my.ini 文件（对于 Windows）：

```
log-bin = /path/to/log-bin/file
binlog-format=row
lower_case_table_names=1
```

MySQL 5 中的 mysqldump 默认情况下将备份所有触发器，但不备份存储过程/函数。有两个控制备份的 mysqldump 参数如下：

- routines（默认为 FALSE，备份存储过程）；
- triggers（默认为 TRUE，备份触发器）。

为了确保存储/备份中包含 dotCMS 存储过程，必须将-routines 参数传递给 mysqldump，命令如下：

```
#mysqldump  --routines > outputfile.sql
```

8.2.2　RESTful API 管理索引

dotCMS 使用 RESTful API 来管理网站搜索索引，也可以用 CURL 通过命令行来管理索引。

在以下所有命令行示例中，将“localhost:8080”替换为你的 dotCMS 实例域名或端口，并提供 YOURINDEXNAME（你的索引名字）、YOURINDEX（你的索引）、YOURPATH（你的路径）或 DESIRED_REPLICAS_NUMBER（期望的副本数量）的实际值。

在 dotCMS 后端上，单击“站点搜索”|“索引”选项卡上的“刷新”按钮，可以查看 CURL 命令对索引的更改，而不是刷新浏览器。

我们可以创建一个新的索引。以下命令将创建一个新的索引，默认情况下，它将被命名为 sitesearch_[timestamp]。

```
#curl
http://localhost:8080/DotAjaxDirector/com.dotmarketing.sitesearch.
ajax.SiteSearchAjaxAction/u/admin@dotcms.com/p/admin/cmd/createSiteSear
chIndex/shards/2
```

提供一个索引别名，命令如下：

```
#curl
http://localhost:8080/DotAjaxDirector/com.dotmarketing.sitesearch.
ajax.SiteSearchAjaxAction/u/admin@dotcms.com/p/admin/cmd/createSiteSear
chIndex/shards/2/alias/YOURINDEXALIAS
```

这样将会创建一个具有指定别名的网站搜索索引。

通过以下命令将把索引下载到一个文件中。

```
#curl
http://localhost:8080/DotAjaxDirector/com.dotmarketing.sitesearch.
ajax.SiteSearchAjaxAction/u/admin@dotcms.com/p/admin/cmd/downloadIndex/
indexName/YOURINDEXNAME/ > INDEXNAME.zip
```

这里将索引作为 JSON 压缩文件进行下载。

也可以根据别名下载索引。命令如下：

```
#curl
http://localhost:8080/DotAjaxDirector/com.dotmarketing.sitesearch.
ajax.SiteSearchAjaxAction/u/admin@dotcms.com/p/admin/cmd/downloadIndex/
indexAlias/YOURINDEXALIAS/ > INDEXALIAS.zip
```

还可以恢复索引。如果指定了 aliasToRestore，则恢复到关联的索引：

```
#curl -F aliasToRestore=torestore -F clearBeforeRestore=true -F
uploadedfiles[]=@sitesearch_20120611161645.zip http://localhost:8080/
DotAjaxDirector/com.dotmarketing.sitesearch.ajax. SiteSearchAjaxAction/
u/admin@dotcms.com/p/admin/cmd/restoreIndex
```

如果没有指定 indexToRestore 或 aliasToRestore 选项，则通过默认索引完成恢复：

```
#curl -F uploadedfiles[]=@sitesearch_20120611161645.zip http://localhost:
8080/DotAjaxDirector/com. dotmarketing.sitesearch.ajax.
SiteSearchAjaxAction/u/admin@dotcms.com/p/admin/cmd/restoreIndex
AjaxAction/u/admin@dotcms.com/p/admin/cmd/restoreIndex
```

需要注意的是：“还原索引”的 CURL 命令是异步发生的，并且在索引被上传之后而且被还原到 ElasticSearch 之前将会返回。要查看正在恢复的索引进度，可以刷新索引列表屏幕。

清理索引，命令如下：

```
#curl
http://localhost:8080/DotAjaxDirector/com.dotmarketing.sitesearch.
ajax.SiteSearchAjaxAction/u/admin@dotcms.com/p/admin/cmd/clearIndex/
indexName/YOURINDEXNAME
```

激活索引，命令如下：

```
#curl
http://localhost:8080/DotAjaxDirector/com.dotmarketing.sitesearch.
ajax.SiteSearchAjaxAction/u/admin@dotcms.com/p/admin/cmd/activateIndex/
indexName/YOURINDEXNAME
```

禁用索引，命令如下：

```
#curl http://localhost:8080/DotAjaxDirector/com.dotmarketing.
sitesearch.ajax.SiteSearchAjaxAction/u/admin@dotcms.com/p/admin/cmd/
deactivateIndex/indexName/YOURINDEXNAME
```

列出所有非活跃的索引，命令如下：

```
#curl http://localhost:8080/DotAjaxDirector/com.dotmarketing.
sitesearch.ajax.SiteSearchAjaxAction/u/admin@dotcms.com/p/admin/
cmd/getNotActiveIndexNames
```

删除索引，命令如下：

```
#curl http://localhost:8080/DotAjaxDirector/com.dotmarketing.
sitesearch.ajax.SiteSearchAjaxAction/u/admin@dotcms.com/p/admin/
cmd/deleteIndex/indexName/YOURINDEXNAME
```

获取索引名称，命令如下：

```
#curl http://localhost:8080/DotAjaxDirector/com.dotmarketing.
sitesearch.ajax.SiteSearchAjaxAction/u/admin@dotcms.com/p/admin/
cmd/getIndexName/indexAlias/YOURINDEXALIAS
```

更新副本，命令如下：

```
#curl http://localhost:8080/DotAjaxDirector/com.dotmarketing.sitesearch.
ajax.SiteSearchAjaxAction/u/admin@dotcms.com/p/admin/cmd/updateReplicas/
indexName/YOURINDEXNAME/replicas/DESIRED_REPLICAS_NUMBER
```

删除搜索作业，命令如下：

```
#curl http://localhost:8080/DotAjaxDirector/com.dotmarketing.sitesearch.
ajax.SiteSearchAjaxAction/u/admin@dotcms.com/p/admin/cmd/deleteJob/
taskName/YOURTASKNAME
```

8.2.3　自动客服机器人

浏览公司网站的潜在客户有时需要在线客服提供一对一的解答和服务。为了节省人力成本，可以开发自动客服机器人，回答一些简单和常见的问题。

HTTP 协议中的通信只能由客户端发起。使用 WS 或者 WSS 协议的 WebSocket 允许服务器端发起通信。WebSocket 是 HTML 5 开始提供的一种浏览器与服务器间进行全双工通信的网络技术。依靠这种技术可以实现客户端和服务器端的长连接，双向实时通信。

WebSocket 的工作流程是：浏览器通过 JavaScript 向服务端发出建立 WebSocket 连接的请求，在 WebSocket 连接建立成功后，客户端和服务端就可以通过 TCP 连接传输数据。

服务器端采用 Java 实现，可以运行在支持 WebScoket 的 Web 服务器中，这里是 Tomcat。而客户端则是运行在浏览器中包含 JavaScript 调用的 HTML 网页，只有一些新的浏览器支持 WebScoket，如 Chrome 或者 FireFox。

Java 服务器端代码如下：

```
/**
 * 通过@ServerEndpoint 注解指定客户端访问的 URL 地址
 */
@ServerEndpoint("/Chat/{who}")
public class Chat {
/**
 * 连接建立成功调用的方法
 * @param session  可选的参数。需要通过 Session 来给客户端发送数据
 */
@OnOpen
public void onMessage(@PathParam("who") String who, Session session) {
    push("欢迎与机器人对话", session);
}

    /**
     * 收到客户端消息后调用这个方法
     * @param message 客户端发送过来的消息
     * @param session 可选的参数
```

```
     */
@OnMessage
public void onMessage(@PathParam("who") String who, String message,
        Session session) {
        String ans = "hi";
        push("机器人回答: " + ans, session);          //向客户端返回信息
}

/**
     * 发生错误时调用
     * @param session
     * @param error
     */
@OnError
public void onError(Session session,
         java.lang.Throwable throwable){
    System.out.println("client onError executed." +throwable);
}

    /**
     * 关闭连接
     */
    @OnClose
    public void onClose() {
        String message = "has disconnection.";
        System.out.println(message);
    }

    /**
     * 发送消息给客户端
     */
    public void push(String message, Session session) {
        session.getAsyncRemote().sendText(message);
}
}
```

用于测试的客户端 Java 代码如下：

```
@ClientEndpoint
public class WebsocketClientEndpoint {
    Session userSession = null;                    //保存和服务器的会话连接
    private MessageHandler messageHandler;         //消息处理器

    public WebsocketClientEndpoint(URI endpointURI) {
        try {
            WebSocketContainer container =
                ContainerProvider.getWebSocketContainer();
            container.connectToServer(this, endpointURI);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    /**
     *  用于连接打开事件的回调钩子
```

```
     *
     * @param userSession 打开的 userSession
     */
    @OnOpen
    public void onOpen(Session userSession) {
        System.out.println("opening websocket");
        this.userSession = userSession;
    }

    /**
     *用于连接关闭事件的回调钩子
     *
     * @param userSession 将要关闭的 userSession
     * @param reason 连接关闭的原因
     */
    @OnClose
    public void onClose(Session userSession, CloseReason reason) {
        System.out.println("closing websocket");
        this.userSession = null;
    }

    /**
     *消息事件的回调钩子。当一个客户端发送一条消息时，将调用该方法
     *
     * @param message 文本消息
     */
    @OnMessage
    public void onMessage(String message) {
        if (this.messageHandler != null) {
            this.messageHandler.handleMessage(message);
        }
    }

    /**
     *注册消息处理器
     *
     * @param msgHandler
     */
    public void addMessageHandler(MessageHandler msgHandler) {
        this.messageHandler = msgHandler;
    }

    /**
     * 发送消息
     *
     * @param message
     */
    public void sendMessage(String message) {
        this.userSession.getAsyncRemote().sendText(message);
    }

    /**
     *消息处理器
     */
```

```
    public static interface MessageHandler {
        public void handleMessage(String message);
    }
}
```

通过 IP 地址 123.56.152.236 测试客户端。代码如下：

```
String url = "ws://123.56.152.236/bot/chat/luogang"; //连接服务器的 URL 地址
final WebsocketClientEndpoint clientEndPoint =
          new WebsocketClientEndpoint(new URI(url));

//添加监听器
clientEndPoint.addMessageHandler(new
WebsocketClientEndpoint.MessageHandler() {
    public void handleMessage(String message) {
        System.out.println(message);
    }
});

//为了接收从 WebSocket 来的消息，等待 5 秒
Thread.sleep(5000);
```

然后完成实际的网页客户端。先引入 reconnecting-websocket.js：

```
<script language="JavaScript" type="text/javascript" src="reconnecting-
websocket.js"></script>
```

使用 WebSocket 实现和服务器端聊天的网页源代码如下：

```
<body>
  <div id="logs"></div>                              //用于显示对话的历史记录
  <div>
<input id="msg" type="text" placeholder="消息"
          //处理回车符
          onkeydown = "if (event.keyCode == 13) document.getElementById
          ('btnSearch').click()"
             />
    <button id="btnSearch" type="submit" value="talk" onClick="talk()">
    说话</button>
  </div>
  <script>
    var iam = '我';
    var host = location.origin.replace(/^http/, 'ws');   //主机名
    var context = window.location.pathname.substring(0,
    window.location.pathname.indexOf('/', 2));      //路径
    var url = host + context + '/Chat/' + iam;      //服务器端的 WS 地址
    var ws = new ReconnectingWebSocket(url);        //和服务器端建立连接

    ws.onmessage = function (message) {             //处理服务器返回的消息内容
      var tag = document.createElement('p');
      tag.appendChild(document.createTextNode(message.data));
      document.getElementById('logs').appendChild(tag);
    };

    function talk() {                               //向服务器发送消息
```

```
        var msg = document.getElementById('msg').value;
        if (msg) {
            ws.send(msg);                                    //通过WebSocket发送消息
            document.getElementById('msg').value = '';//重置输入框中的消息内容
            var tag = document.createElement('p');
            tag.appendChild(document.createTextNode(iam +':'+msg));
            document.getElementById('logs').appendChild(tag);
                                                             //记录发送的消息历史
        }
    }
  </script>
</body>
```

可以用爬虫抓取问题，然后替换其中的关键词为通用的类别，就能得到问句模板。例如，问句“糖尿病怎么治疗”替换其中的“糖尿病”为疾病名，得到问句模板“<疾病>怎么治疗”，把这样的方法叫做泛化。例如，问句“载人飞船从地球到火星需要多长时间？”泛化成“<交通工具>从<地点>到<地点>需要多长时间？”。

答案句也可以做泛化处理，如问句“多功能监控表的宽范围交/直流通用电源是多少？”对应的答案句“宽范围交直流通用电源：AC/DC 80V~270V”，能泛化成：<num>V~<num>V。

不同意图的问句需要不同的处理方式。例如“1 加 1”和“今天天气怎么样”需要不同的处理器。现有的一些处理器包括：

- 处理简单问答对用的 ChatHandler；
- 处理加法的 AddHandler；
- 处理英文单词的 WordHandler 等。

处理器从问句提取关键词，提取出来的关键词以键/值对的形式存入 PairListString 对象。

```
//键可以重复
public class PairListString {
String[] values;                                             //存储所有名称和值
int count;

public PairListString(int initialCapacity) {
    values = new String[initialCapacity * 2];
}

/**
 * 增加一个名称/值对
 *
 * @param x 名称
 * @param y 名称对应的值
 */
public void addPair(String x, String y) {
    if (count * 2 >= values.length) {
        values = Arrays.copyOf(values, values.length * 2);
    }
    values[count * 2] = x;
```

```
        values[count * 2 + 1] = y;
        count++;
    }

    public String getX(int index) {                          //根据下标得到键
        return values[index * 2];
    }

    public String getY(int index) {                          //根据下标得到值
        return values[index * 2 + 1];
    }

    public String get(String key){                           //得到键对应的值
        for (int i = 0; i < count; ++i) {
            if(values[i * 2].equals(key)){
                return values[i * 2 + 1];
            }
        }

        return null;
    }
    }
```

使用这个类存放从问句“糖尿病怎么治疗”中提取的参数，示例代码如下：

```
PairListString questionArgs = new PairListString(1);

String type = "DiseaseName";                                 //键
String diseaseName = "糖尿病";                               //值

questionArgs.addPair(type, diseaseName);
System.out.println(questionArgs.get(type));                  //输出糖尿病
```

动态加载问句处理器如下：

```
String handleClass = "questionHandler."+handleName+"Handler"; //得到类名
Class<? extends QuestionHandler> clz = Class.forName(handleClass)
        .asSubclass(QuestionHandler.class); //返回 QuestionHandler 的子类
QuestionHandler answer = clz.newInstance();                  //得到问句处理器
String ans = answer.getAnswer(kb,g.questionArgs);            //依据问句处理器返回答案
```

一个问句所得到的规则对象如下：

```
public class Rule {
public ArrayList<String> rhs =
                new ArrayList<String>();                     //右边的 Token 类型序列
public ArrayList<TokenType> lhs =
                new ArrayList<TokenType>();                  //左边的 Token 类型序列
public HashMap<String, HashSet<String>> words = new HashMap<String,
HashSet<String>>();//词表
}
```

处理问句的规则如下：

```
String ruleStr = "<num>{plusnum1}加<num>{plusnum2}";
```

```
ArrayList<RuleToken> tokens = IERuleParser.getSeq(ruleStr,67);//规则及编号
for (RuleToken t : tokens) {
System.out.println(t);                                    //输出规则中的每个 Token
}
Rule rule = RuleBuilder.create(tokens);                   //根据 Token 序列创建规则
System.out.println(rule);                                 //输出生成的问句规则
System.out.println("is valid:"+rule.valid());             //验证问句规则是否有效
```

根据问句模板处理问句：

```
QuestionGrammar g = new QuestionGrammar();                //问句文法库

String right = "<Begin><num>{plusnum1}加<num>{plusnum2}<End>";  //问句模板
g.add("Add", right);                                      //问句意图:Add

right = "<Begin><DiseaseName>{disease}怎么治疗<End>";
g.add("Cure", right);                                     //问句意图:Cure

String type = "DiseaseName";
String diseaseName = "糖尿病";
//仅仅增加词而不是加规则
g.addWord(diseaseName, type);

TextExtractor ie = new TextExtractor(g);                  //问句文法对应的文本提取器

String question = "糖尿病怎么治疗";
AdjList adjList = ie.getLattice(question);                //返回问句对应的词图
System.out.println(adjList);
```

根据问句模板返回答案：

```
KnowlegeBase kb = new KBSqlite();                  //使用 Sqlite 数据库存储知识库
GrammarAnswer answer = new GrammarAnswer(kb);             //根据文法返回答案

String question = "糖尿病怎么治疗";
String ans = answer.getAnswer(question);                  //返回知识库中存储的答案

System.out.println(ans);                                  //输出答案
```

为了能够回答可能出现在语音识别结果中的“What is three plus four?”这类问题，需要识别英文数字。识别英文数字的自动机代码如下：

```
public static Automaton getEnNum() {
    Automaton num = BasicAutomata.makeString("zero");  //接收英文单词
    num = num.union(BasicAutomata.makeString("one"));  //并联英文单词
    num = num.union(BasicAutomata.makeString("two"));
    num = num.union(BasicAutomata.makeString("three"));
    num = num.union(BasicAutomata.makeString("four"));
    num = num.union(BasicAutomata.makeString("five"));
    num = num.union(BasicAutomata.makeString("six"));
    num = num.union(BasicAutomata.makeString("seven"));
    num = num.union(BasicAutomata.makeString("eight"));
```

```
        num = num.union(BasicAutomata.makeString("nine"));
        num = num.union(BasicAutomata.makeString("ten"));
        num = num.union(BasicAutomata.makeString("eleven"));
        num = num.union(BasicAutomata.makeString("twelve"));
        num = num.union(BasicAutomata.makeString("thirteen"));
        num = num.union(BasicAutomata.makeString("fourteen"));
        num = num.union(BasicAutomata.makeString("fifteen"));
        num = num.union(BasicAutomata.makeString("sixteen"));
        num = num.union(BasicAutomata.makeString("seventeen"));
        num = num.union(BasicAutomata.makeString("eighteen"));
        num = num.union(BasicAutomata.makeString("nineteen"));
        num = num.union(BasicAutomata.makeString("twenty"));

        return num;
    }
```

测试识别英文数字：

```
Automaton numAutomaton = AutomatonFactory.getEnNum();
                                                    //得到识别英文数字的自动机
String num="one";                                   //待测试的英文文本
System.out.println(BasicOperations.run(numAutomaton, num));//判断是否能接收
```

增加问句规则：

```
QuestionGrammar g = new QuestionGrammar();      //问句文法

String right = "<Begin>What is <EnNum> plus <EnNum><End>";
g.add("EnAdd", right); // 这个问句规则使用处理器 EnAdd
```

让机器人可以回答“找<人名>的照片”，这里使用 OpenCV 实现人脸识别。

人脸识别是 OpenCV 额外模块的一部分。需要在如下网址下载：

https://github.com/opencv/opencv_contrib。

还有最新版本的 OpenCV，地址如下：

https://opencv.org/releases.html。

使用 OpenCV 之前需要静态加载本地库：

static{ System.loadLibrary(Core.NATIVE_LIBRARY_NAME); }

可以添加一个 JVM 参数来绕过库路径问题：

```
-Djava.library.path=/home/edward/Downloads/opencv-3.3.1/build/lib/
```

训练人脸识别器（在这里是从检测到/裁剪的人脸图像目录），代码如下：

```
ArrayList<Mat> sourceImages = new ArrayList<>();   //源图像
List<String> namesIndexList = new ArrayList<>();   //名字列表
List<Integer> namesIntList = new ArrayList<>();    //人名编号
Files.list(path).forEach(file -> {
    String filename = file.getFileName().toString();
if (filename.contains("-") && filename.endsWith("jpg")){
        //从文件名中提取人名，例如从 edd-1.jpg 中提取 edd
        String personName = filename.substring(0, filename.indexOf("-"));
        if (!namesIndexList.contains(personName)){
```

```
            namesIndexList.add(personName);
        }
        Mat image = Imgcodecs.imread(file.toString());
        Imgproc.cvtColor(image, image, Imgproc.COLOR_BGR2GRAY);
                                                    //将图像转换为灰度图
        sourceImages.add(image);
        namesIntList.add(namesIndexList.indexOf(personName));//加入人名编号
    }
});
FaceRecognizer faceRecognizer = LBPHFaceRecognizer.create();
                                                    //使用 LBP 直方图实现人脸识别
MatOfInt matOfInt = new MatOfInt();
matOfInt.fromList(namesIntList);
faceRecognizer.train(sourceImages, matOfInt); //训练数据集
```

这里值得注意的是，在训练时需要传递一个 Mat 图像的数组和一个表示每个人脸标识整数的 Mat 对象。将每个新名称添加到一个 ArrayList 中，这样人名就有了一个编号，然后将该编号添加到列表中。如果有多个人脸照片，则编号会重复放入列表。文件命名约定是任意的 edd-1.jpg、edd2.jpg 和 ben-1.jpg 等。

预测过程很简单，如下：

```
Imgproc.cvtColor(image, image, Imgproc.COLOR_BGR2GRAY);//将图像转换为灰度图
faceRecognizer.predict_label(image);          //返回识别出来的用户编号
```

可以用返回的整数查找人名。

8.3　搜索文档

本节使用 Elasticsearch 搜索公开的药物临床试验项目信息。首先使用网络爬虫抓取网页和 Word 文档中的信息，然后索引和搜索数据。

8.3.1　爬虫抓取信息

使用 PhantomJS 抓取网站：

```
System.setProperty("phantomjs.binary.path",
    "F:/crawler/phantomjs-2.1.1-windows/bin/phantomjs.exe");

String path = "F:/crawler/";                    //保存抓取结果的路径
int pgNo = 1;                                   //页码号
WebDriver driver = new PhantomJSDriver();

String url = "http://www.chinadrugtrials.org.cn/eap/clinicaltrials.
searchlist";
driver.get(url);
Thread.sleep(2000);                             //等待一段时间，让网页下载完成
```

```
while (true) {
    String content = driver.getPageSource();

    writeToFile(content, path + "trials/" + pgNo + ".html", "utf-8");
                                                    //写入本地文件中

    // 查找下一页链接
    By by = By.className("page_next");

    WebElement nextPage = driver.findElement(by);
    if(nextPage ==null) break;                      //如果没有下一页了则退出
    nextPage.click();
    Thread.sleep(2000);
    pgNo++;
}
driver.quit();
```

把下载的网页中的信息写入 Sqlite 数据库。以下用 Java 程序展示如何连接到现有的数据库。如果这个数据库不存在，就会即时创建它，并最终返回一个数据库对象。

```
Class.forName("org.sqlite.JDBC");                   //加载数据库驱动程序
String path_to_sqlite_db = "./db/trials.db";        //指定数据库文件
Connection conn = DriverManager.getConnection("jdbc:sqlite:"+path_to_
sqlite_db);                                         //建立连接
```

首先用程序自动创建数据库文件和表：

```
//建立连接。如果数据库文件不存在，则自动创建这个文件
Connection conn = getConnect();

Statement stmt = conn.createStatement();
//创建表。如果表不存在，则自动创建这个表，否则跳过这一步
String sql = "create table if not exists trials (regno string, status
string)";

stmt.executeUpdate(sql);
stmt.close();
conn.close();
```

然后把网页中的数据放入数据库：

```
String inSql = "insert into trials (regno ,status)values(?,?)";
                                                    //插入数据库的 SQL 语句
PreparedStatement insertStmt = conn
        .prepareStatement(inSql);

String path = "F:/crawler/trials";
File file = new File(path);
String htmls[] = file.list();                                   //遍历文件

for (int j = 0; j < htmls.length; j++) {
    File f = new File(path + "/" + htmls[j]);
    String content = FileUtils.readFileToString(f, "utf8");
                                                                //读取文件内容
    Document doc = org.jsoup.Jsoup.parse(content);
```

```
                                                    //使用 Jsoup 解析网页格式的文本
    Element es1 = doc.getElementsByClass("Tab").first();   //读取表格

    Elements tr = es1.getElementsByTag("tr");

    for (int i = 1; i < tr.size(); ++i) {
        Elements td = tr.get(i).getElementsByTag("td");

        String regno = td.get(1).text();                   //得到注册号
        String status = td.get(2).text();                  //得到状态

        insertStmt.setString(1, regno);                    //写入注册号
        insertStmt.setString(2, status);                   //写入状态
        insertStmt.executeUpdate();                        //执行插入语句
    }
}

insertStmt.close();                                        //关闭语句
```

使用 FireFox 自动下载 Word 文件。在 pom.xml 中引入依赖：

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.8.1</version>
</dependency>

<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-server</artifactId>
    <version>3.8.1</version>
</dependency>
```

首先从 https://github.com/mozilla/geckodriver 下载 Windows 下运行的驱动文件 geckodriver.exe，然后使用 Selenium 驱动的 FireFox 下载单个 doc 文件。代码如下：

```
//指定 FireFox 驱动所在的路径
System.setProperty("webdriver.gecko.driver",
"D:/driver_win32/geckodriver.exe");

FirefoxOptions firefoxOptions = new FirefoxOptions();

//设置自动保存 doc 文件的配置信息
FirefoxProfile firefoxProfile = new FirefoxProfile();
String downPath = "c:\\downloads";
firefoxProfile.setPreference("browser.download.folderList",2);
firefoxProfile.setPreference("browser.download.manager.showWhenStarting
",false);
firefoxProfile.setPreference("browser.download.dir",downPath);
//直接保存到本地硬盘
firefoxProfile.setPreference("browser.helperApps.neverAsk.saveToDisk","
application/msword");
//如果是自动保存 PDF 文件，则使用如下的配置信息
```

```
//firefoxProfile.setPreference("browser.helperApps.neverAsk.saveToDisk"
, "application/pdf");

firefoxOptions.setProfile(firefoxProfile);
WebDriver driver = new FirefoxDriver(firefoxOptions);

String url = "file:///d:\\crawler\\detail\\CTR20171478 详细信息.html";
                                                    //从指定的网址下载
driver.get(url);
Thread.sleep(2000);                                 //等待 2 秒

By by = By.className("download");                   //找到“下载”按钮
WebElement down = driver.findElement(by);
down.click();                                       //自动单击“下载”按钮

Thread.sleep(5000);                                 //等待 5 秒

driver.quit();                                      //退出 FireFox

File src = new File(downPath+"/详细信息.doc");
File dest = new File("d:/crawler/doc/详细信息.doc");
FileUtils.moveFile(src, dest);                      //转移文件
```

遍历所有的网页文件，下载对应的 Word 文档：

```
File file = new File("d:/crawler/detail/");
File[] fileList = file.listFiles();

for (File f : fileList) {                           //遍历网页存放路径下的文件
        String docFile = f.getName().substring(0, f.getName().length() - 5)
            + ".doc";                               //根据编号生成 Word 文件名
    File dest = new File("d:/crawler/doc/" + docFile);
    if(dest.exists()){                              //如果已经存在，则不重复下载
        continue;
    }
    //执行下载

    FileUtils.moveFile(src, dest);                  //转移下载的文件
}
```

使用 Aspose.Words 从下载的 Word 文档中提取信息。首先定义存放信息的实体类，表示药物临床试验信息的 Trials.cs 文件内容如下：

```
public class Trials{
    public string regno;                            //登记号
    public string sponsor;                          //申办者联系人
    public string tel;                              //电话
    public string email;                            //邮箱
    public string address;                          //地址
    public string pubdate;                          //公示日期
    public string company;                          //申办者联系人
    public string testcaseno;                       //试验方案编号
```

```
    public string acceptanceno;                    //临床申请受理号
    public string drugtype ;                       //药物类型
    public string fundingsource;                   //试验项目经费来源
    public string purpose;                         //试验目的
    public string testtype;                        //试验分类
    public string teststage;                       //试验分期
    public string designtype;                      //设计类型
    public string random;                          //随机化
    public string blind;                           //法盲
    public string scope;                           //试验范围
    public string subjectage;                      //受试者年龄
    public string subjectsex;                      //受试者性别
    public string healthysubject;                  //健康受试者
    public string includecriteria;                 //入选标准
    public string exclusioncriteria;               //排除标准
    public string targetenrollmentnumber;          //目标入组人数
    public string actualenrollednumber;            //实际入组人数
    public string testdrug;                        //试验药
    public string controlleddrug;                  //对照药
    public string mainendpointindicator_evaluationtime;
                                                   //主要终点指标及评价时间
    public string secondaryendpointindicator_evaluationtime;
                                                   //次要终点指标及评价时间
    public string datasecuritycommission;          //数据安全监察委员会（DMC）
    public string injuryinsurance;                 //为受试者购买试验伤害保险
    public string firstcasedate;                   //第一例受试者入组日期
    public string principalresearchername;         //主要研究者姓名
    public string principalresearcherphone;        //主要研究者电话
    public string principalresearcheremail;        //主要研究者 E-mail
    public string participatingagency;             //各参加机构信息
    public string ethicscommittee;                 //伦理委员会信息
}
```

找出文档中所有的表格：

```
Aspose.Words.Document oleDocument = new Aspose.Words.Document
(docFileName);
NodeCollection tables = oleDocument.GetChildNodes(NodeType.Table, true);
```

遍历表格：

```
Trials trials = new Trials();                      //创建存放提取信息的实体类
foreach (Table table in tables){
                    //遍历表格中的每行
                IEnumerator rowEnumerator = table.Rows.GetEnumerator();

                while(rowEnumerator.MoveNext()) //移动到下一行
                {
                    Row row = (Row)rowEnumerator.Current;  //得到当前行
                    //遍历一行中的每个单元格
                    IEnumerator cellEnumerator = row.Cells.
```

```
                GetEnumerator();

                while(cellEnumerator.MoveNext()){  //移动到下一个单元格
                    Cell cell = (Cell)cellEnumerator.Current;
                                                   //得到当前单元格

                    string text = cell.GetText();
                                                   //得到当前单元格中的文本

                    if(text.Equals("联系人姓名")){
                        cellEnumerator.MoveNext();
                        cell = (Cell)cellEnumerator.Current;
                        text = cell.GetText();      //得到联系人姓名
                        trials.sponsor = text.Substring(0,text.
                        Length-1);
                    }
                    else if(text.Equals("联系人电话")){
                        cellEnumerator.MoveNext();
                        cell = (Cell)cellEnumerator.Current;
                        text = cell.GetText();      //得到联系人电话
                        trials.tel = text.Substring(0,text.Length-1);
                    }
                    else if(text.Equals("联系人 Email")){
                        cellEnumerator.MoveNext();
                        cell = (Cell)cellEnumerator.Current;
                        text = cell.GetText();      //得到联系人 E-mail
                        trials.email = text.Substring(0,text.
                        Length-1);
                    }
                    else if(text.Equals("联系人邮政地址")){
                        cellEnumerator.MoveNext();
                        cell = (Cell)cellEnumerator.Current;
                        text = cell.GetText(); //得到联系人邮政地址
                        trials.address = text.Substring(0,text.
                        Length-1);
                    }
                }

            }
}
```

从多行表格中提取信息：

```
public static string getMutiLine(NodeCollection tables,string label){
                                                   //输入提取标签
        String content ="";                        //提取内容
        foreach (Table table in tables){           //遍历每个表格

            IEnumerator rowEnumerator = table.Rows.GetEnumerator();
                                                   //得到行迭代器
            while(rowEnumerator.MoveNext())        //移动到下一行
            {
                Row row = (Row)rowEnumerator.Current;  //得到当前行
```

```
                    //得到单元格迭代器
                    IEnumerator cellEnumerator = row.Cells.GetEnumerator();

                    while(cellEnumerator.MoveNext()){
                        Cell cell = (Cell)cellEnumerator.Current;

                        string text = cell.GetText();

                        if(text.Equals(label)){
                            cellEnumerator.MoveNext();
                            cell = (Cell)cellEnumerator.Current;
                            text = cell.GetText();

                            while (true){

                                if(Regex.IsMatch(text, @"^\d+")){
                                                    //判断是否以数字开头
                                    content +=text.Substring(0,text.
                                    Length-1)+"\r\n";

                                    rowEnumerator.MoveNext();
                                                    //移动到下一行
                                    row = (Row)rowEnumerator.Current;

                                    cellEnumerator = row.Cells.
                                    GetEnumerator();
                                    cellEnumerator.MoveNext();
                                    cell = (Cell)cellEnumerator.Current;
                                    text = cell.GetText();
                                }
                                else{
                                    break;
                                }
                            }
                            break;
                        }
                    }
                }
            }

            return content;
}
```

更新 Sqlite 数据库中的数据，把从 Word 文档中提取出来的信息存入数据库：

```
public static void save(Trials trials){
            //与数据库建立连接
        string dbFile = "F:/workspace/DrugCrawlerPhantomJ/db/trials.db";
        SQLiteConnection m_dbConnection =
            new SQLiteConnection("Data Source="+dbFile+";Version=3;");
        m_dbConnection.Open();

         string sql =
     "update trials set tel=?,sponsor=?,pubdate=?,company=?,testcaseno=?,
includecriteria=?,exclusioncriteria=?,targetenrollmentnumber=?,actualen
```

```
rollednumber=?,testdrug=?,controlleddrug=?,mainendpointindicator_evalua
tiontime=?,secondaryendpointindicator_evaluationtime=?,datasecuritycomm
ission=?,injuryinsurance=?,firstcasedate=?,principalresearchername=?,pr
incipalresearcherphone=?,principalresearcheremail=?,participatingagency
=?,ethicscommittee=? where regno=?";                    //更新表的 SQL 语句
        SQLiteCommand command = new SQLiteCommand(sql, m_dbConnection);
        //设置参数
        SQLiteParameter tel = new SQLiteParameter();
        SQLiteParameter sponsor = new SQLiteParameter();
        SQLiteParameter regno = new SQLiteParameter();
        SQLiteParameter pubdate = new SQLiteParameter();
        SQLiteParameter company = new SQLiteParameter();
        SQLiteParameter testcaseno = new SQLiteParameter();
        SQLiteParameter includecriteria = new SQLiteParameter();
        SQLiteParameter exclusioncriteria = new SQLiteParameter();
        SQLiteParameter targetenrollmentnumber = new SQLiteParameter();

        SQLiteParameter actualenrollednumber = new SQLiteParameter();
        SQLiteParameter testdrug = new SQLiteParameter();
        SQLiteParameter controlleddrug = new SQLiteParameter();
        SQLiteParameter mainendpointindicator_evaluationtime = new
SQLiteParameter();
        SQLiteParameter secondaryendpointindicator_evaluationtime = new
SQLiteParameter();
        SQLiteParameter datasecuritycommission = new SQLiteParameter();
        SQLiteParameter injuryinsurance = new SQLiteParameter();
        SQLiteParameter firstcasedate = new SQLiteParameter();
        SQLiteParameter principalresearchername = new SQLiteParameter();
        SQLiteParameter principalresearcherphone = new SQLiteParameter();
        SQLiteParameter principalresearcheremail = new SQLiteParameter();
        SQLiteParameter participatingagency = new SQLiteParameter();
        SQLiteParameter ethicscommittee = new SQLiteParameter();

        //设置参数
        command.Parameters.Add(tel);
        command.Parameters.Add(sponsor);
        command.Parameters.Add(pubdate);
        command.Parameters.Add(company);
        command.Parameters.Add(testcaseno);
        command.Parameters.Add(includecriteria);
        command.Parameters.Add(exclusioncriteria);
        command.Parameters.Add(targetenrollmentnumber);
        command.Parameters.Add(actualenrollednumber);
        command.Parameters.Add(testdrug);
        command.Parameters.Add(controlleddrug);
        command.Parameters.Add(mainendpointindicator_evaluationtime);
        command.Parameters.Add(secondaryendpointindicator_
        evaluationtime);
        command.Parameters.Add(datasecuritycommission);
        command.Parameters.Add(injuryinsurance);
        command.Parameters.Add(firstcasedate);
        command.Parameters.Add(principalresearchername);
        command.Parameters.Add(principalresearcherphone);
        command.Parameters.Add(principalresearcheremail);
        command.Parameters.Add(participatingagency);
```

```
            command.Parameters.Add(ethicscommittee);
            command.Parameters.Add(regno);

            //设置值
            tel.Value = trials.tel;
            sponsor.Value = trials.sponsor;
            regno.Value = trials.regno;
            pubdate.Value = trials.pubdate;
            company.Value = trials.company;
            estcaseno.Value = trials.testcaseno;
            includecriteria.Value = trials.includecriteria;
            exclusioncriteria.Value = trials.exclusioncriteria;
            targetenrollmentnumber.Value = trials.targetenrollmentnumber;
            actualenrollednumber.Value = trials.actualenrollednumber;
            testdrug.Value = trials.testdrug;
            controlleddrug.Value = trials.controlleddrug;
            mainendpointindicator_evaluationtime.Value = trials.
            mainendpointindicator_evaluationtime;
            secondaryendpointindicator_evaluationtime.Value =
                    trials.secondaryendpointindicator_evaluationtime;
            datasecuritycommission.Value = trials.datasecuritycommission;
            injuryinsurance.Value = trials.injuryinsurance;
            firstcasedate.Value = trials.firstcasedate;
            principalresearchername.Value = trials.principalresearchername;
            principalresearcherphone.Value = trials.principalresearcherphone;
            principalresearcheremail.Value = trials.principalresearcheremail;
            particpatingagency.Value = trials.particpatingagency;
            ethicscommittee.Value = trials.ethicscommittee;

            command.ExecuteNonQuery();

            m_dbConnection.Close();
    }
```

8.3.2　在 Linux 下使用.NET

我们可以使用以下命令在 CentOS 7 上安装.NET Core：

```
# yum install centos-release-dotnet
# yum install rh-dotnet20
```

使用.NET Core：

```
# scl enable rh-dotnet20 bash
# dotnet --info
```

运行例子如下：

```
# mkdir helloworld && cd helloworld
# dotnet new console
# dotnet run
```

代码是用模板构建的。这里使用的是控制台应用程序模板，并且该项目的依赖关系由 dotnet restore 命令自动检索，可从 https://www.nuget.org 网站上提取。

如果查看目录，会看到创建了以下文件：

```
Program.cs
helloworld.csproj
```

其中，Program.cs 是 C＃控制台应用程序代码。helloworld.csproj 是 MSBuild 兼容的项目文件。

dotnet run 这个命令做了两件事：首先其建立了代码文件，并运行新建的代码文件。无论何时执行 dotnet run 命令，它都将检查* .csproj 文件是否已被更改，并将运行 dotnet restore 命令。其次，dotnet run 命令还可以检查是否有任何源代码已被修改，并在后台运行 dotnet build 命令构建可执行文件，并运行可执行文件。

除了 dotnet，在 Linux 下还可以使用 Mono 开发 C#应用。Mono 是 C#语言的开源跨平台开发系统，它也是可执行运行时引擎的名称，支持运行由 MCS 编译器生成的输出文件。而 MCS 是 Mono 系统的编译器。mcs 命令编译出的 IL 代码可以用 mono 命令执行。

可以用 make 命令构建 Mono 项目。构建模板如下：

```
#将其更改为 Main 类文件的名称，不带文件扩展名
MAIN_FILE = App/Program

#将其更改为项目文件夹的深度
#如果需要，还可以为通用项目文件夹添加前缀
CSHARP_SOURCE_FILES = $(wildcard */*/*.cs */*.cs *.cs)

#添加需要的标志到 compilerCSHARP_FLAGS = -out:$(EXECUTABLE)
CSHARP_FLAGS = -out:$(EXECUTABLE)

#更改为环境中的编译器
CSHARP_COMPILER = mcs

#如果需要，更改可执行文件
EXECUTABLE = $(MAIN_FILE).exe

#如果需要，可根据用户的操作系统更改删除命令
RM_CMD = -rm -f $(EXECUTABLE)

all: $(EXECUTABLE)

$(EXECUTABLE): $(CSHARP_SOURCE_FILES)
    @ $(CSHARP_COMPILER) $(CSHARP_SOURCE_FILES) $(CSHARP_FLAGS)
    @ echo compiling...

run: all
    ./$(EXECUTABLE)

clean:
    @ $(RM_CMD)

remake:
    @ $(MAKE) clean
    @ $(MAKE)
```

8.3.3　NEST 客户端

Elasticsearch.NET 是一个非常底层的客户端。NEST 是一个在 Elasticsearch.NET 上构建的高层.NET 客户端，可以从 https://github.com/elastic/elasticsearch-net 下载最新的版本。

可以通过单个节点或者指定多个节点使用连接池连接到 Elasticsearch 集群。连接到单个节点的例子如下：

```
var node = new Uri("http://myserver:9200"); //创建连接到 Elasticsearch 的 URI
var settings = new ConnectionSettings(node);  //连接设置
var client = new ElasticClient(settings);  //创建 ElasticClient 客户端
```

通过连接池连接的例子如下：

```
var nodes = new Uri[]
{
    new Uri("http://myserver1:9200"),
    new Uri("http://myserver2:9200"),
    new Uri("http://myserver3:9200")
};

var pool = new StaticConnectionPool(nodes);
var settings = new ConnectionSettings(pool);
var client = new ElasticClient(settings);
```

使用连接池要比单个节点连接到 Elasticsearch 更有优势，如支持负载均衡、故障转移等。可以输出集群的状态：

```
var res = client.Raw.ClusterHealth();
Console.WriteLine(res.SuccessOrKnownError);
```

然后在连接设置中指定默认索引：

```
var settings = new ConnectionSettings(
    node,
    defaultIndex: "trails"                              //设置默认索引为 trails
);
```

连接参数配置到 app.config 文件中。

```
<add key="Search-Uri" value="http://localhost:9200" />
```

可以使用这样的代码配置客户端：

```
var node = new Uri(ConfigurationManager.AppSettings["Search-Uri"]);
                                                        //读配置文件
var settings = new ConnectionSettings(node);            //建立连接设置
settings.PrettyJson(); // 为了调试，可以输出设置的 JSON 格式
var client = new ElasticClient(settings);
```

ElasticClient.CreateIndex 方法创建索引，代码如下：

```
var descriptor = new CreateIndexDescriptor("trials")
    .Settings(s => s.NumberOfShards(5).NumberOfReplicas(1)); // 5 个主分片
client.CreateIndex(descriptor);
```

如果删除索引，可以使用 DeleteIndexDescriptor。例如：

```
var descriptor = new DeleteIndexDescriptor("trials").Index("trials");
client.DeleteIndex(descriptor);
```

可以使用基于属性的映射来定义索引结构。在 NEST 中有两个映射属性可用，即 ElasticType 和 ElasticProperty。

- ElasticType 属性定义 Elastic 索引类型，可以指定 Name 和 IdProperty 选项，其中 Name 提供类型名称，IdProperty 指定哪个属性应该用做 ID 键。
- ElasticProperty 属性指定模型属性应如何编制索引。

对于每个属性，我们可以分配以下 4 个选项：Name、Type、Index 和 Analyzer。Name 选项指定在 Elastic 索引中调用的属性；Type 选项指定如何存储属性（Elastic 内部类型有 string、integer / long、float / double、boolean 和 null）；Index 选项具有以下值。

- Analyzed：在索引之前分析和处理属性的值；
- NotAnalyzed：属性的值不会被分析，并将被编入索引；
- No：属性的值不会建立索引。

在 TrialsProject.cs 文件中定义临床信息的索引结构，代码如下：

```
[ElasticType(IdProperty = "Id", Name = "trials_project")]
public class TrialsProject
{
   [ElasticProperty(Name = "_id", Index = FieldIndexOption.NotAnalyzed,
Type = FieldType.String)]
   public Guid? Id { get; set; }                    //Id 列

   [ElasticProperty(Name = "title", Index = FieldIndexOption.Analyzed, Type
   = FieldType.String)]
   public string Title { get; set; }                //临床项目名称

   [ElasticProperty(Name = "body", Index = FieldIndexOption.Analyzed, Type
   = FieldType.String)]
   public string Body { get; set; }                 //临床项目内容描述

   public override string ToString()
   {
      return string.Format("Id: '{0}', Title: '{1}', Body: '{2}'", Id,
      Title, Body);
   }
}
```

使用属性中的类型和映射信息来创建索引。如下：

```
var res = client.CreateIndex(ci => ci
   .Index("trials")
   .AddMapping<TrialsProject>(m => m.MapFromAttributes()));
Console.WriteLine(res.RequestInformation.Success);
```

可以单条或者批量插入数据。插入单条记录的方法如下：

```
var trials = new Trials
{
```

```
    Id = 1,
    Title = "甲磺酸沙芬酰胺片人体生物等效性研究",
    Body = "生物等效性试验/生物利用度试验"
};

//也可以通过 settings.DefaultIndex 指定索引
var response = client.Index(trials, idx => idx.Index("trials"));
Console.WriteLine(response.RequestInformation.Success);//看是否成功索引
```

Query DSL 是一个通用的查询框架。可以通过 Elasticsearch.Net API 向搜索服务器发送 JSON 格式的 Query DSL。

基本词查询：

```
String keyWords = "二甲双胍";  //查询词
var rs = client.Search<Trials>(s => s.Index("trials").QueryString
(keyWords));
Console.WriteLine(JsonConvert.SerializeObject(rs.Documents));
```

查询标题列：

```
var res = client.Search(s => s
.Query(q => q
.Match(m => m.OnField(f => f.Title).Query("二甲双胍"))));//指定查询列
Console.WriteLine(res.RequestInformation.Success);
//查询结果需要返回一个结果总数，用于知道可以翻多少页
Console.WriteLine(res.Hits.Count());                          //输出返回结果数量

//遍历每条查询结果
foreach (var hit in res.Hits)
{
   Console.WriteLine(hit.Source);                             //输出命中结果
}
```

如果需要文档包括所有的词，可以使用 AND 连接查询词。如下：

```
var res = client.Search<Trials>(s => s
  .Query(q => q
  .Match(m => m
   .OnField(f => f.Title)
   .Query("二甲双胍")
   .Operator(Operator.And))));                                //指定查询操作符
```

在查询输入中需要设置两个参数：从第几个结果开始返回文档，以及最多返回多少个文档。为了把所有的文档都找出来，可以使用 MatchAll()函数查询。如下：

```
var res = client.Search<Trials>(s => s
   .From(0)                                                   //指定开始行
   .Size(5)                                                   //指定结束行
   .Query(q => q.MatchAll());
```

使用 Should()函数同时搜索标题和正文：

```
var boolRes2 = client.Search<Trials>(s => s
  .Query(q => q
     .Bool(b => b                                            //布尔查询
```

```
            .Should(sh =>
                sh.Match(mt1 => mt1.OnField(f1 => f1.Title).Query("二甲双胍")) ||
                sh.Match(mt2 => mt2.OnField(f2 => f2.Body).Query("对照"))
    )))
    .Sort(o => o.OnField(p => p.Title).Ascending()));     //标题升序
```

然后增加一个 must_not 约束条件：

```
var boolRes3 = client.Search<Trials>(s => s
  .Query(q => q
     .Bool(b => b
        .Should(sh =>
           sh.Match(mt1 => mt1.OnField(f1 => f1.Title).Query("二甲双胍")) ||
           sh.Match(mt2 => mt2.OnField(f2 => f2.Title).Query("编号")))
        .Must(ms =>                                          //必须满足的条件
           ms.Match(mt2 => mt2.OnField(f => f.Body).Query("对照")))
        .MustNot(mn =>                                       //排除的词
           mn.Match(mt2 => mt2.OnField(f => f.Body).Query("终止"))
  )))
  .Sort(o => o.OnField(p => p.Title).Ascending()));     //标题升序
```

可以在多个列上设置高亮。首先指定哪些列需要高亮，然后在命中结果中得到高亮段。可以通过一个 Dictionary 类的实例指定要高亮的多个列。代码如下：

```
new SearchRequest<Project>
{
    Query = new MatchQuery
    {
        Query = "二甲双胍",
        Field = "name.standard"
    },
    Highlight = new Highlight
    {
        PreTags = new[] { "<tag1>" },              //高亮标签
        PostTags = new[] { "</tag1>" },
        Fields = new Dictionary<Field, IHighlightField>
        {
            { "name.standard", new HighlightField
                {
                    Type = HighlighterType.Plain, //使用基本的高亮器
                    ForceSource = true,
                    FragmentSize = 150,            //高亮段大小
                    NumberOfFragments = 3,         //段的数量
                    NoMatchSize = 150              //没有匹配词时返回的文本长度
                }
            },
            { "leadDeveloper.firstName", new HighlightField
                {
                    Type = "fvh",                  //使用快速向量高亮器
                    BoundaryMaxScan = 50,
                    PreTags = new[] { "<name>"},
                    PostTags = new[] { "</name>"},
                    HighlightQuery = new MatchQuery
```

```
                {
                    Field = "leadDeveloper.firstName",
                    Query = "Kurt Edgardo Naomi Dariana Justice Felton"
                }
            }
        },
        { "state.offsets", new HighlightField
            {
                Type = HighlighterType.Postings,  //使用文档列表高亮器
                PreTags = new[] { "<state>"},
                PostTags = new[] { "</state>"},
                HighlightQuery = new TermsQuery
                {
                    Field = "state.offsets",
                    Terms = new [] { "stable", "bellyup" }
                }
            }
        }
    }
}
```

8.4　本章小结

本章介绍了双语句对搜索、内容管理系统站内检索和药物临床试验项目信息文档搜索的 Elasticsearch 应用案例。

在双语句对搜索案例中，可以借助双语句对和 PanLex（网址是 https://panlex.org/）这样的多语言词典挖掘更多的词语上下文关系，提升分词准确度。

使用 Elasticsearch 实现站内检索的 dotCMS 初始版本发布于 2005 年。在 PHP 版本的内容管理系统流行的时代，dotCMS 是 Java 的一个可选项。早期的 dotCMS 直接使用 Lucene 实现搜索功能，借助内容中的双语句对，让多语言的网站能够让搜索结果更准确。

在最后介绍的药物临床试验项目信息的抓取和搜索案例中，搜索界面运行于 Mono 平台。

Mono 是由 Microsoft 的子公司 Xamarin 和.NET Foundation 领导的免费开源项目，旨在创建符合 ECMA 标准的兼容.NET Framework 的工具集，其中包括 C＃编译器和公共语言运行平台。Mono 可以使.NET 应用程序运行在 Windows、Linux 和 Mac OS 等不同平台上。

Mono 项目在开放源代码社区中一直存在争议，因为它实现了微软专利所涵盖的.NET Framework 的部分功能。继 Microsoft 自 2014 年开始开源多项核心.NET 技术，以及于 2016 年初收购 Xamarin 之后，Mono 项目获得了更新的专利承诺。

参 考 文 献

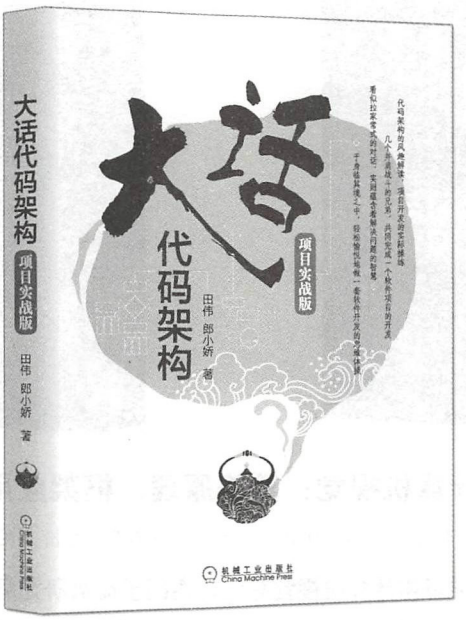
[1] 罗刚，张子宪，崔智杰．Java 中文文本信息处理——从海量到精准[M]．北京：清华大学出版社，2017．

[2] 罗刚．解密搜索引擎技术实战——Lucene&Java 精华版（第 3 版）[M]．北京：电子工业出版社，2016．

[3] 罗刚．网络爬虫全解析——技术、原理与实践 [M]．北京：电子工业出版社，2017．

[4] 尼克．人工智能简史 [M]．北京：人民邮电出版社，2017．

推荐阅读



大话代码架构（项目实战版）

作者：田伟 郎小娇　书号：978-7-111-57701-0　定价：69.00元

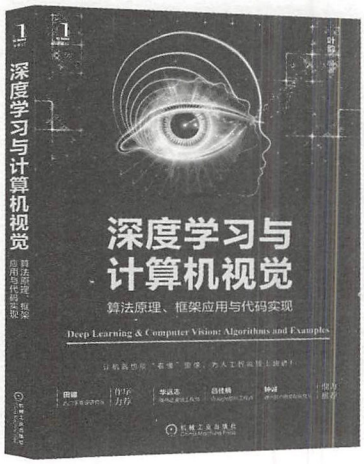
代码架构的风趣解读，项目开发的实际操练
几个并肩战斗的兄弟，共同完成一个软件项目的开发
看似拉家常式的对话，实则蕴含着解决问题的智慧
于身临其境之中，轻松愉悦地做一套软件开发的思维体操

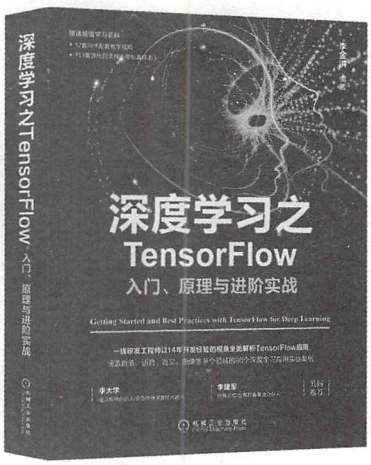
本书以一个真实的项目案例——“晋商卡”从无到有的整个开发过程为主线，用大话的语言风格，风趣幽默地讲解了代码架构的相关知识。本书通过5个人物角色，模拟实际的项目开发过程，以对话形式抛出问题，然后解决问题，让你在身临其境中轻松愉快地掌握代码架构的知识。

本书涵盖的主要内容有敏捷开发的方法论、项目开发流程、传统的三层架构、源代码管理、几种常见的实体关系模型、使用IoC和接口、使用缓存和静态页面减少服务器压力、在项目中使用消息队列、尝试使用前端框架、微信公众号开发及小程序开发。

本书适合对代码架构感兴趣的初学者和爱好者阅读。另外，高校学生和参加软件开发的培训学员也可将本书作为兴趣读物。对于初入职场还比较迷茫的程序员，本书可以作为一本提高读物来阅读。建议阅读本书的读者具有一定的C#语言基础。

推荐阅读





深度学习与计算机视觉：算法原理、框架应用与代码实现

作者：叶韵　书号：978-7-111-57367-8　定价：79.00元

全面、深入剖析深度学习和计算机视觉算法，西门子高级研究员田疆博士作序力荐！

Google软件工程师吕佳楠、英伟达高级工程师华远志、理光软件研究院研究员钟诚博士力荐！

本书全面介绍了深度学习及计算机视觉中的基础知识，并结合常见的应用场景和大量实例带领读者进入丰富多彩的计算机视觉领域。作为一本“原理+实践”教程，本书在讲解原理的基础上，通过有趣的实例带领读者一步步亲自动手，不断提高动手能力，而不是枯燥和深奥原理的堆砌。

本书适合对人工智能、机器学习、深度学习和计算机视觉感兴趣的读者阅读。阅读本书要求读者具备一定的数学基础和基本的编程能力，并需要读者了解Linux的基本使用。

深度学习之TensorFlow：入门、原理与进阶实战

作者：李金洪　书号：978-7-111-59005-7　定价：99.00元

磁云科技创始人/京东终身荣誉技术顾问李大学、创客总部/创客共赢基金合伙人李建军共同推荐

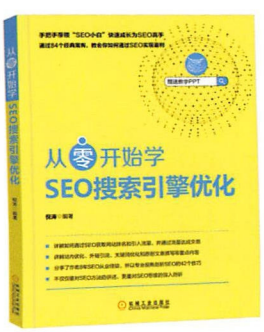
一线研发工程师以14年开发经验的视角全面解析TensorFlow应用

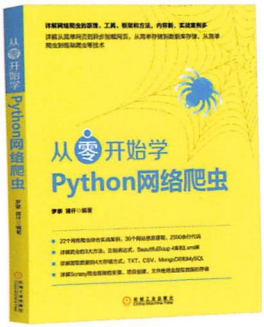
涵盖数值、语音、语义、图像等多个领域的96个深度学习应用实战案例！

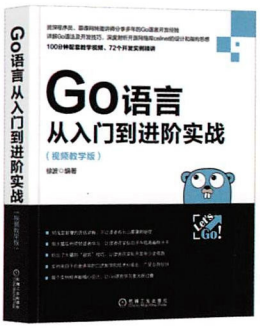
本书采用“理论+实践”的形式编写，通过大量的实例（共96个），全面而深入地讲解了深度学习神经网络原理和TensorFlow使用方法两方面的内容。书中的实例具有很强的实用性，如对图片分类、制作一个简单的聊天机器人、进行图像识别等。书中每章都配有一段教学视频，视频和图书的重点内容对应，能帮助读者快速地掌握该章的重点内容。本书还免费提供了所有实例的源代码及数据样本，这不仅方便了读者学习，而且也能为读者以后的工作提供便利。

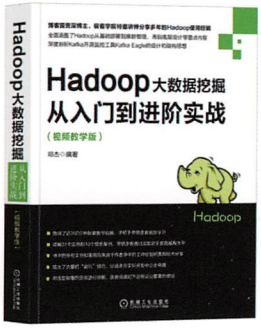
本书特别适合TensorFlow深度学习的初学者和进阶读者作为自学教程阅读。另外，本书也适合作为相关培训学校的教材，以及各大院校相关专业的教学参考书。

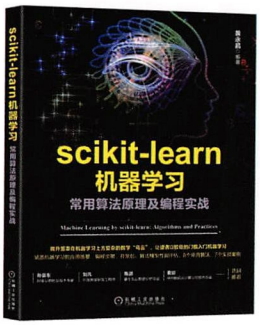
推荐阅读

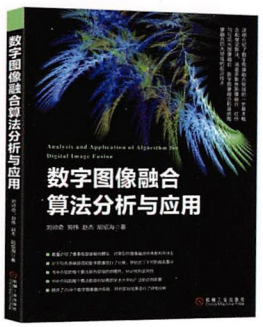












欢迎IT领域的各位技术人员洽谈出书事宜。如果您有写书或投稿意向，请加QQ或微信与编辑具体商谈。

QQ：627173439

微信：oyzx_sp

Elasticsearch
搜索引擎开发实战

本书精华内容

- 基于中文分词的中文搜索算法
- 基于字词混合索引的搜索算法
- 英文分词算法
- 英文句子切分算法
- Word2vec实现算法
- 人脸识别融合
- CURL爬虫算法
- OkHttp爬虫
- EM算法实现词对齐
- CRC32算法检验文件完整性
- Netty通信框架分析
- Zen发现机制
- Spring Boot MVC开发Web应用
- Vue.js开发前端应用
- Elasticsearch生成JSON串
- 双语句对搜索案例
- 内容管理系统站内检索案例
- 药物临床试验项目信息爬虫案例

业内专家点评

随着人工智能的发展，大数据搜索和自然语言处理等技术也开始受到开发人员的重视。而Elasticsearch以其强大的分布式搜索和可视化功能成为了这个领域的主流开发工具。本书作者长期深耕爬虫、搜索和数据处理等相关领域，有深厚的技术底蕴，他们编写的这本书便是对自己开发经验的一个总结，值得相关开发者借鉴。

—— 众包产品经理 **张运成**

虽然百度、搜狗和360几家公司占据了搜索的主要市场份额，但各大电商、社交、人工智能和区块链等互联网项目中依然需要用到搜索技术。精通搜索技术的程序员依然深受各大IT公司的欢迎。作者的这本书可以帮助大家掌握Elasticsearch大数据搜索引擎技术，非常适合从事数据挖掘、推荐算法、搜索效能优化的人阅读。另外，书中的一些代码来源于实际项目案例，有很高的参考价值，也推荐给同行参考。

—— 健趣网创始人/CEO **程振宇**

配套资源获取方式

本书源文件等配套资源需要读者下载。请在www.hzbook.com网站上搜索到本书，然后单击“资料下载”按钮即可进入本书页面，找到“配书资源”链接即可下载。

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/ 搜索引擎

ISBN 978-7-111-60348-1

9 787111 603481 >

定价：69.00元